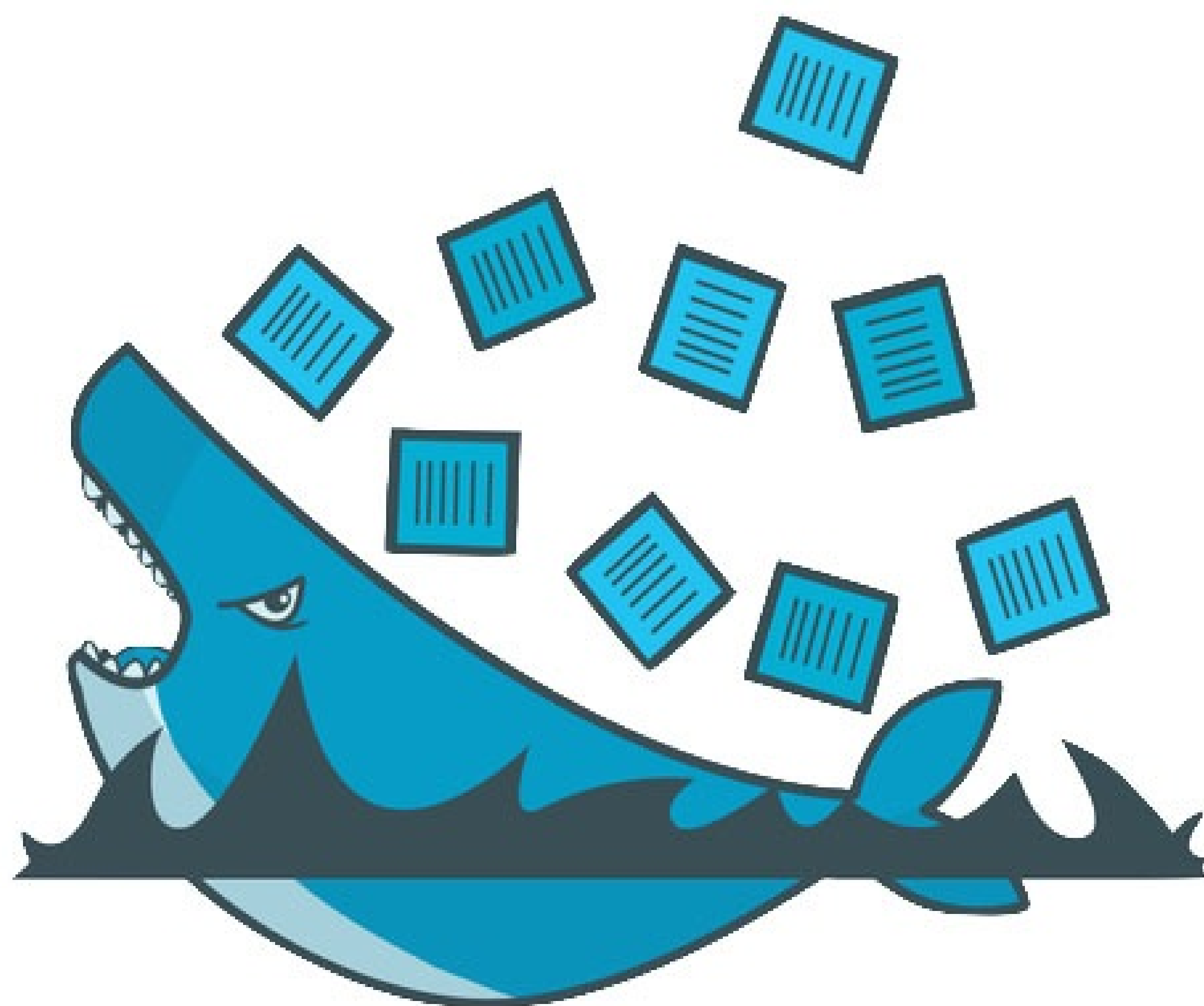
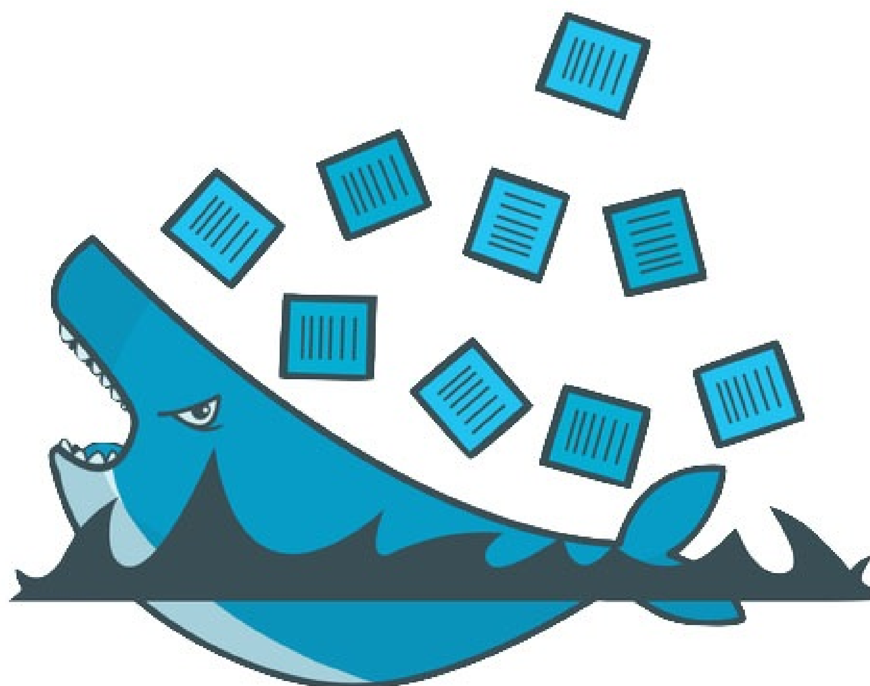
Docker 简明教程



目錄

前言

Docker 简明教程



Docker是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

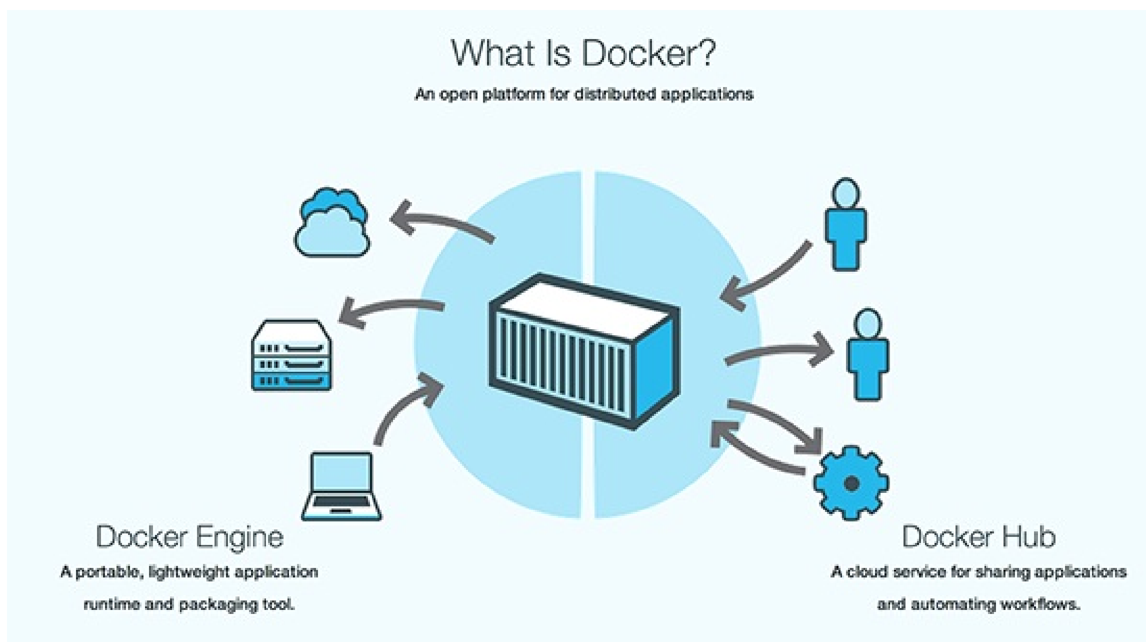
容器是完全使用沙箱机制，相互之间不会有任何接口（类似 iPhone 的 app）。 几乎没有性能开销,可以很容易地在机器和数据中心中运行。最重要的是,他们不依赖于任何语言、框架包括系统。

Docker 简介

1、由PaaS到Container

2013年2月，前Gluster的CEO Ben Golub 和 dotCloud 的 CEO Solomon Hykes 坐在一起聊天时，Solomon谈到想把 dotCloud 内部使用的Container容器技术单独拿出来开源，然后围绕这个技术开一家新公司提供技术支持。28岁点Solomon在使用python开发dotCloud的PaaS云时发现，使用LXC（Linux Container）技术可以打破产品发布过程中应用工程师和系统工程师两者之间无法轻松协作发布产品的难题。这个Container容器技术可以把开发者从日常部署的繁杂工作中解脱出来，让开发者能专心写好程序；从系统工程师到角度来看也是一样的，他们迫切需要从各种混乱的部署中解脱出来，让系统工程师专注在应用的水平扩展、稳定发布的解决方案上。他们深入交谈，觉得这是一次云技术的变革，紧接着在2013年3月Docker0.1发布，拉开来基于云计算平台发布产品方式的变革序幕。

2、Docker 简介

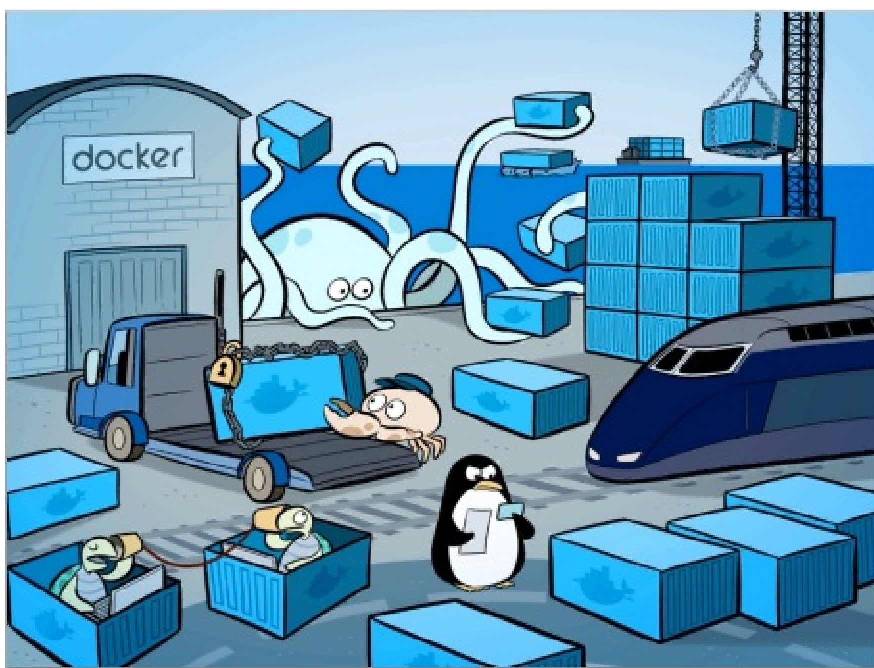


Docker时Docker.Inc公司开源的一个基于LXC技术之上搭建的Container容器引擎，源代码托管在Github上，基于Go语言并遵从Apache2.0协议开源。Docker在2014年6月召开DockerConf2014技术大会吸引了IBM、Google、RedHat等业界知名公司的关注和技术支持，无论是从Github上到代码活跃度，还是RedHat宣布REHL7

中正式支持Docker，都给业界一个信号，这是一项创新的技术解决方案。就连Google公司的Computer Engine 也支持Docker在其中之上运行，国内BAT先锋企业百度 Baidu App Engine（BAE）平台也是以 Docker作为PasS云基础。

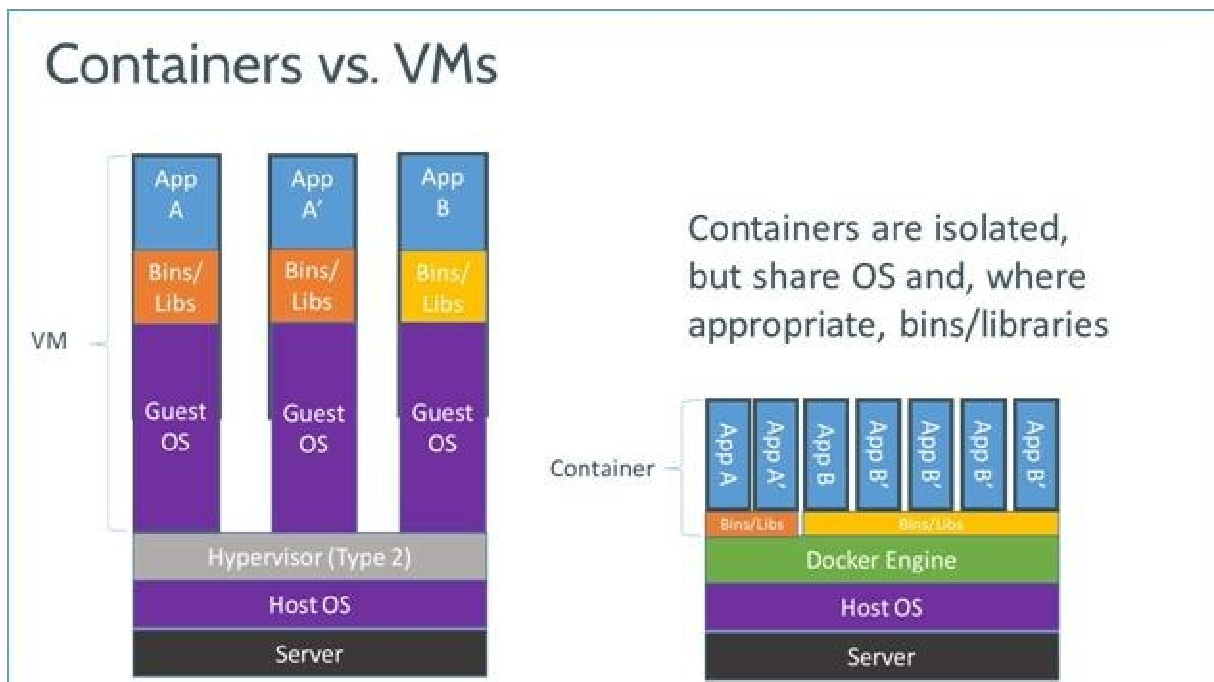
3、Docker产生的目的就是要解决以下问题：

1）环境管理复杂：从各种OS到各种中间件再到各种App，一款产品能够成功发布，作为开发者需要关心的东西太多，且难于管理，这个问题在软件行业中普遍存在并需要直接面对。Docker可以简化部署多种应用实例工作，比如Web应用、后台应用、数据库应用、大数据应用比如Hadoop集群、消息队列等等都可以打包成一个image部署。如下图所示：



2）云时代的到来：AWS的成功，引到开发者将应用转移到云上，解决来硬件管理的问题，然而软件配置和管理香瓜的问题依然存在。Docker的出现正好能帮助软件开发者开阔思路，尝试新的软件管理方法来解决这个问题。

3）虚拟化手段的变化：云时代采用标配硬件来降低成本，采用虚拟化手段来满足用户按需分配的资源需求以及保证可用性和隔离性。然而无论是KVM还是Xen，在Docker卡来都是在浪费资源，因为用户需要的是高校运行环境而非OS，GuestOS即浪费资源，又难于管理，更加轻量级大LXC更佳灵活和快速：



4）**LXC**的便携性: LXC在 Linux 2.6 的 Kernel 里就已经存在了，但是其设计之初并非为云计算考虑的，缺少标准化的描述手段和容器的可便携性，决定其构建出的环境难于分发和标准化管理(相对于KVM之类image和snapshot的概念)。Docker就在这个问题上做出了实质性的创新方法。

Docker 核心技术

Docker核心是一个操作系统级虚拟化方法, 理解起来可能并不像VM那样直观。我们从虚拟化方法的四个方面：

1. 隔离性 Namespace
2. 可配额/可度量 Cgroups
3. 便携性 AUFS
4. 安全性 AppArmor、SELinux、GRSEC

接下来将详细介绍Docker的技术细节。

隔离性 Linux namespace

每个用户实例之间相互隔离, 互不影响。 一般的硬件虚拟化方法给出的方法是VM，而LXC给出的方法是container，更细一点讲就是kernel namespace。其中pid、net、ipc、mnt、uts、user等namespace将container的进程、网络、消息、文件系统、UTS("UNIX Time-sharing System")和用户空间隔离开。

1. pid namespace

   不同用户的进程就是通过pid namespace隔离开的，且不同 namespace 中可以有相同pid。所有的LXC进程在docker中的父进程为docker进程，每个lxc进程具有不同的namespace。同时由于允许嵌套，因此可以很方便的实现 Docker in Docker。

2. net namespace

   有了 pid namespace, 每个namespace中的pid能够相互隔离，但是网络端口还是共享host的端口。网络隔离是通过net namespace实现的， 每个net namespace有独立的 network devices, IP addresses, IP routing tables, /proc/net 目录。这样每个container的网络就能隔离开来。docker默认采用veth的方式将container中的虚拟网卡同host上的一个docker bridge: docker0连接在一起。

3. ipc namespace

   container中进程交互还是采用linux常见的进程间交互方法(interprocess communication - IPC), 包括常见的信号量、消息队列和共享内存。然而同 VM 不同的是，container 的进程间交互实际上还是host上具有相同pid namespace 中的进程间交互，因此需要在IPC资源申请时加入namespace信息 - 每个IPC资源有一个唯一的 32 位 ID。

4. mnt namespace

   类似chroot，将一个进程放到一个特定的目录执行。mnt namespace允许不同namespace的进程看到的文件结构不同，这样每个 namespace 中的进程所看到的文件目录就被隔离开了。同chroot不同，每个namespace中的container在/proc/mounts的信息只包含所在namespace的mount point。

5. uts namespace

UTS("UNIX Time-sharing System") namespace允许每个container拥有独立的hostname和domain name, 使其在网络上可以被视作一个独立的节点而非Host上的一个进程。

6. user namespace

   每个container可以有不同的 user 和 group id, 也就是说可以在container内部用container内部的用户执行程序而非Host上的用户。

控制组 - Control Groups (cgroups) 可配额、可度量

cgroups 实现了对资源的配额和度量。 cgroups 的使用非常简单，提供类似文件的接口，在 /cgroup目录下新建一个文件夹即可新建一个group，在此文件夹中新建task文件，并将pid写入该文件，即可实现对该进程的资源控制。groups可以限制blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net_cls、ns九大子系统的资源，以下是每个子系统的详细说明：

- blkio 这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及usb等等。
- cpu 这个子系统使用调度程序为cgroup任务提供cpu的访问。
- cpuacct 产生cgroup任务的cpu资源报告。
- cpuset 如果是多核心的cpu，这个子系统会为cgroup任务分配单独的cpu和内存。
- devices 允许或拒绝cgroup任务对设备的访问。
- freezer 暂停和恢复cgroup任务。
- memory 设置每个cgroup的内存限制以及产生内存资源报告。
- net_cls 标记每个网络包以供cgroup方便使用。
- ns 名称空间子系统。

以上九个子系统之间也存在着一定的关系.详情请参阅官方文档。

便携性: AUFS

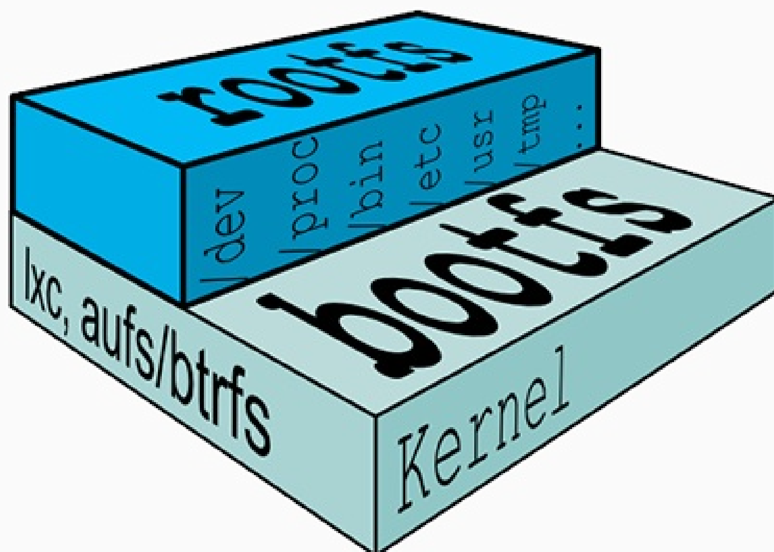
AUFS (AnotherUnionFS) 是一种 Union FS。

简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统, 更进一步的理解, AUFS支持为每一个成员目录(类似Git Branch)设定readonly、readwrite 和 whiteout-able 权限, 同时 AUFS 里有一个类似分层的概念, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。

通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个disk挂到同一个目录下, 另一个更常用的就是将一个 readonly 的 branch 和一个 writeable 的 branch 联合在一起，Live CD正是基于此方法可以允许在 OS image 不变的基础上允许用户在其上进行一些写操作。

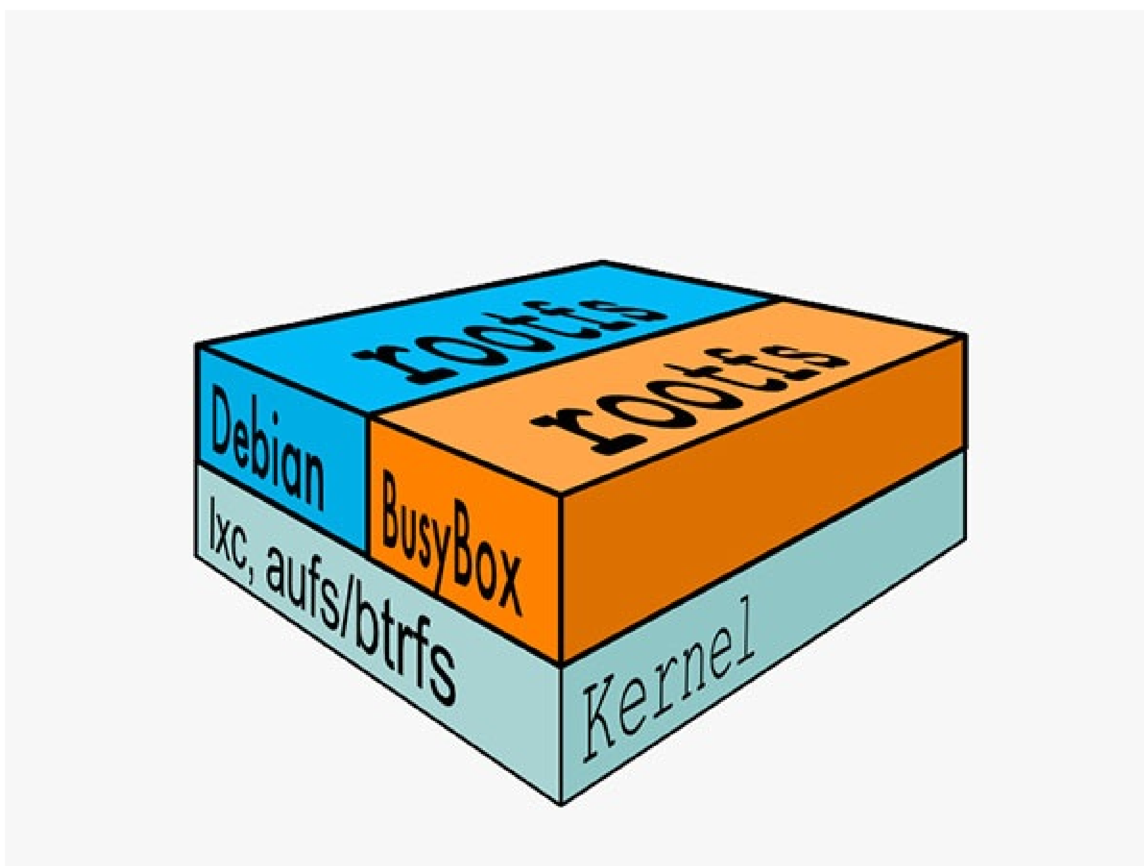
Docker 在 AUFS 上构建的 container image 也正是如此，接下来我们从启动 container 中的 linux 为例来介绍 docker 对AUFS特性的运用。

典型的启动Linux运行需要两个FS: bootfs + rootfs:

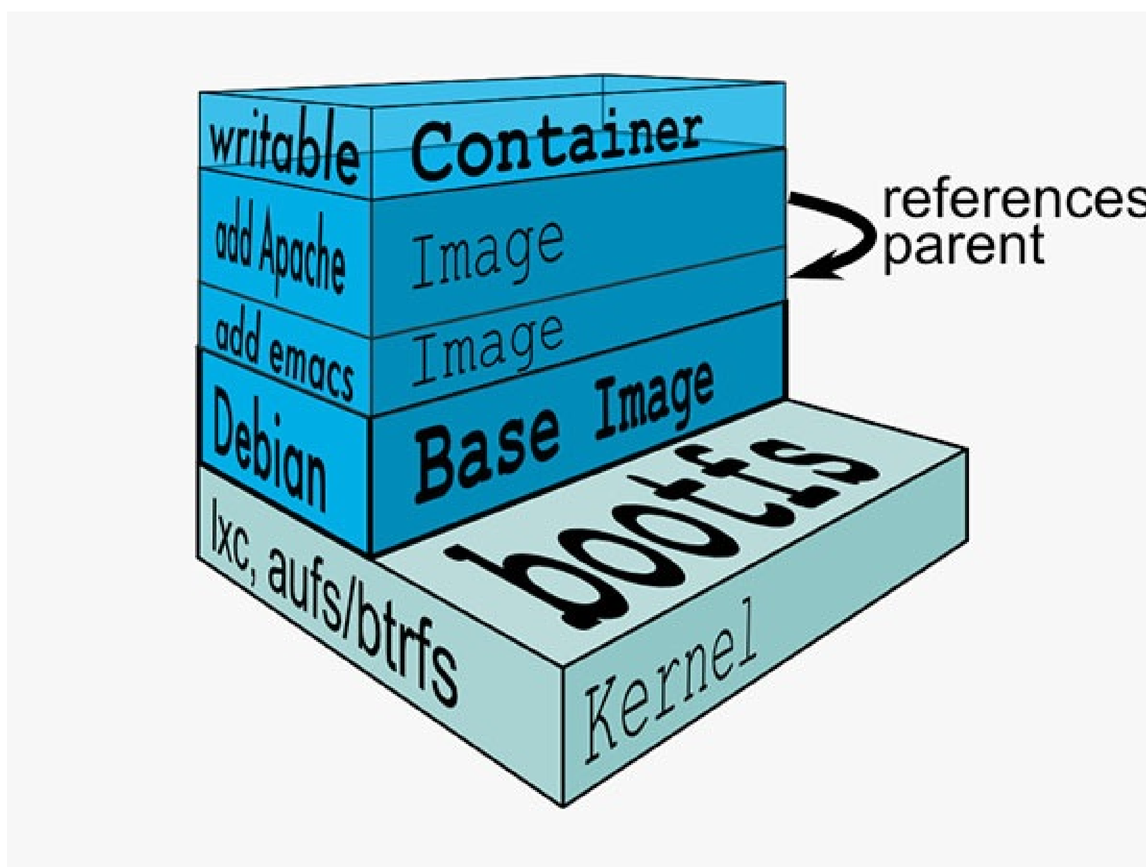


bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了. rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc,/bin, /etc 等标准目录和文件。

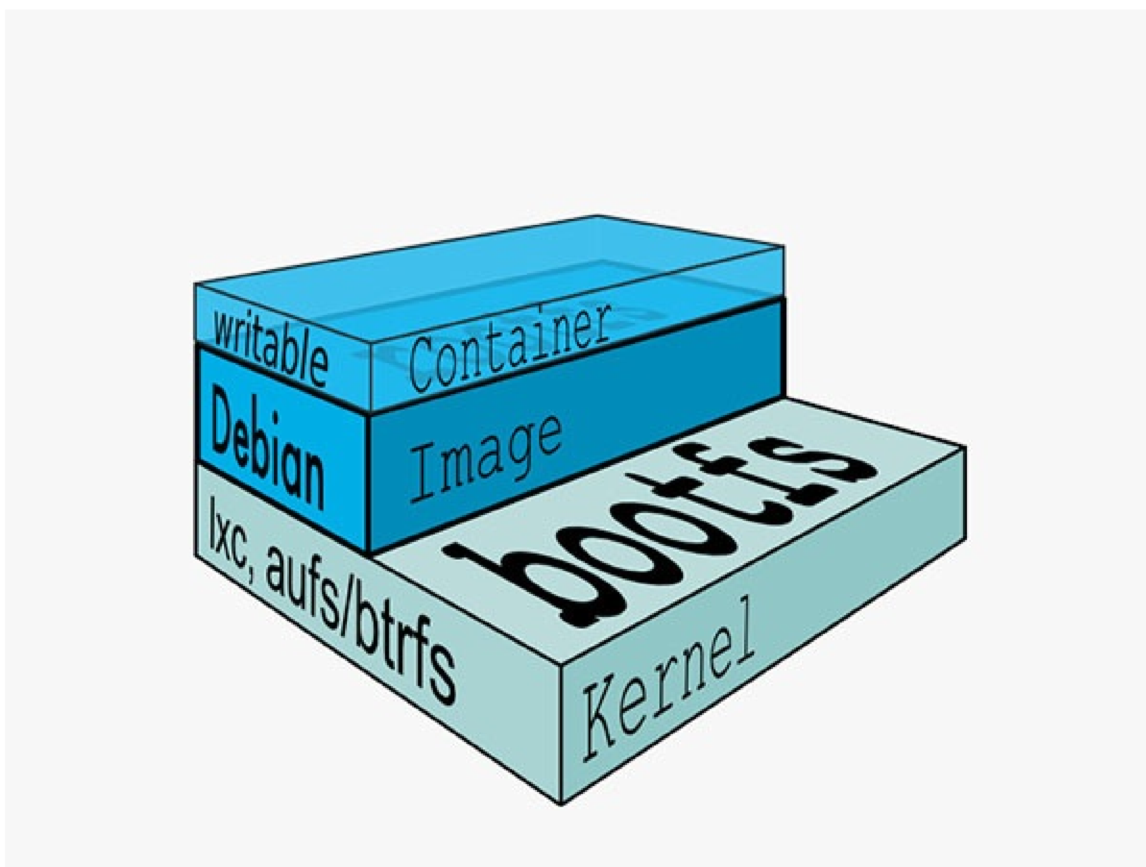
对于不同的linux发行版, bootfs基本是一致的, 但rootfs会有差别, 因此不同的发行版可以公用bootfs 如下图:



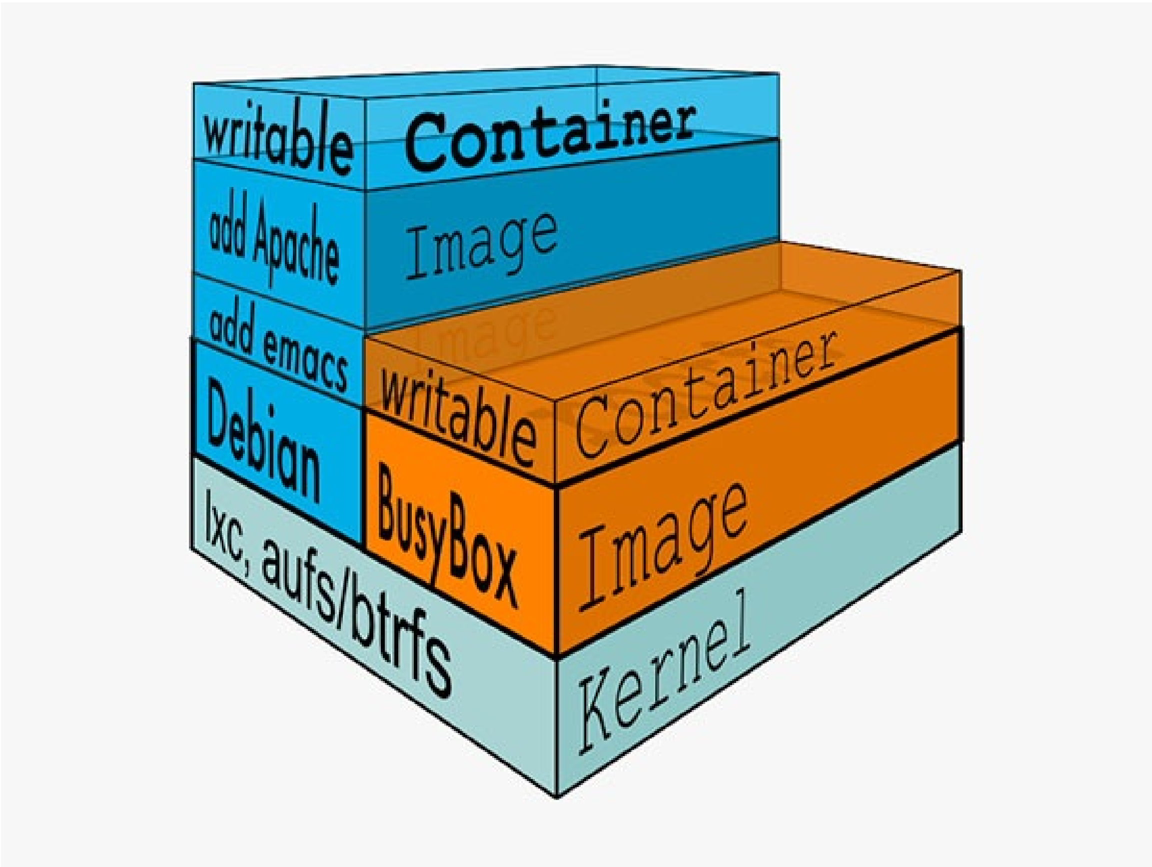
典型的Linux在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 "readwrite" 供用户使用。在Docker中，初始化时也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加, 这样一组readonly和一个writeable的结构构成一个container的运行时态, 每一个FS被称作一个FS层。如下图:



得益于AUFS的特性, 每一个对readonly层文件/目录的修改都只会存在于上层的writeable层中。这样由于不存在竞争, 多个container可以共享readonly的FS层。 所以Docker将readonly的FS层称作 "image" - 对于container而言整个rootfs都是read-write的，但事实上所有的修改都写入最上层的writeable层中, image不保存用户状态，只用于模板、新建和复制使用。



上层的image依赖下层的image，因此Docker中把下层的image称作父image，没有父image的image称作base image。因此想要从一个image启动一个container，Docker会先加载这个image和依赖的父images以及base image，用户的进程运行在writeable的layer中。所有parent image中的数据信息以及 ID、网络和lxc管理的资源限制等具体container的配置，构成一个Docker概念上的container。如下图:


writable
Container
add Apache
Image
add emacs
writable
Container
Debian
BusyBox
Image
lxc, aufs/btrfs
Kernel

安全性: AppArmor, SELinux, GRSEC

安全永远是相对的，这里有三个方面可以考虑Docker的安全特性:

1. 由kernel namespaces和cgroups实现的Linux系统固有的安全标准;
2. Docker Deamon的安全接口;
3. Linux本身的安全加固解决方案,类如AppArmor, SELinux;

由于安全属于非常具体的技术，这里不在赘述，请直接参阅Docker官方文档。

Docker 快速入门

本章节着重介绍Docker的具体使用，文章内容参照 官方文档 、网络资源结合自己实践经验整理撰写而成，若其中有解释不清楚，或者原文翻译问题出错，请及时与我联系纠正。 本章节分为四个部分：

1. Docker安装

   该部分介绍Docker在Centos7试验环境下的安装方法。

2. Docker镜像

   该部分主要介绍 Docker利用Dockerfile来构建构建镜像文件。通常Dcokerfile构建镜像时需要基础镜像，在文章的开始部分介绍构建基础镜像的官方方法。随后，详细讲解Dockerfile的文件结构，以及文件内元素的具体使用方法。

3. Docker容器

   该部分介绍容器的一些基础使用。利用简单的docker命令行来创建docker容器，并且在容器中安装应用等。简单介绍docker命令行的基础使用方法，调试方法等。介绍如何管理容器内产生的数据并将其共享和持久化，同时容器通信的管理实现也做来相关介绍。

4. Docker基本指令和用法

   为了方便查阅，作者采用字典方式来整理docker命令行说明。

安装 Docker (操作系统为CentOS7 64位)

1).采用yum源的方法安装

```
$ sudo yum install docker
```

＊安装过程中会有多次选择y OR n，选择y以继续安装

安装成功后，会在最后几行有提示：

```
Installed:
  docker.x86_64 0:1.7.1-108.el7.centos

Dependency Installed:
  docker-selinux.x86_64 0:1.7.1-108.el7.centos

Complete!
```

2).启动Docker服务

```
$ sudo service docker start
```

或者

```
$ sudo systemctl start docker
```

3).设置开机启动

```
$ sudo  systemctl enable docker
```

4).检查Docker信息

Docker 版本信息：

```
$ sudo docker version
Client version: 1.7.1
Client API version: 1.19
Package Version (client): docker-1.7.1-108.el7.centos.x86_64
Go version (client): go1.4.2
Git commit (client): 3043001/1.7.1
OS/Arch (client): linux/amd64
Server version: 1.7.1
Server API version: 1.19
Package Version (server): docker-1.7.1-108.el7.centos.x86_64
Go version (server): go1.4.2
Git commit (server): 3043001/1.7.1
OS/Arch (server): linux/amd64
$...
```

查看系统(Docker)层面信息，包括管理的images, containers数：

```
$ sudo docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-8:3-28326-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 307.2 MB
 Data Space Total: 107.4 GB
 Data Space Available: 45.61 GB
 Metadata Space Used: 729.1 kB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.147 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/meta
 Library Version: 1.02.93-RHEL7 (2015-01-28)
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.10.0-229.el7.x86_64
Operating System: CentOS Linux 7 (Core)
CPUs: 2
Total Memory: 1.466 GiB
Name: localhost.localdomain
ID: PRVB:3SDE:YL4E:JT5P:5BIR:BUC5:PHXI:HG4B:P753:Y2BI:U7OU:YPGC
$...
```

至此 Docker 在CentOS下安装步骤就完成了！

镜像

镜像是 Docker 的三大组件之一。

Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地，Docker 会从镜像仓库下载（默认是 Docker Hub 公共注册服务器中的仓库），我们也可以搭建一个本地的镜像仓库，但这不是本文的重点。本文将以镜像为中心介绍：

- 如何构建基础镜像
- Dockerfile的基本结构以及详解
- 利用Dockerfile构建镜像

构建基础镜像

我将应用打包到镜像中形成我们所需的镜像，往往需要一个基础的镜像作为我们应用服务的外部环境，那么问题来了，基础镜像从何而来？官方推荐的是直接从官网仓库pull一个，但由于官网被墙的比较厉害，所以这里介绍一些官方提供以及个人方法。

1.使用Debootstrap来创建Ubuntu的base image

```
$ sudo debootstrap raring raring > /dev/null
$ sudo tar -C raring -c . | docker import - raring
a29c15f1bf7a
$ docker run raring cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=13.04
DISTRIB_CODENAME=raring
DISTRIB_DESCRIPTION="Ubuntu 13.04"
```

在docker github上有更多有关基础镜像的介绍

- BusyBox
- CentOS / Scientific Linux CERN (SLC) on Debian/Ubuntu or on CentOS/RHEL/SLC/etc.
- Debian / Ubuntu

2.使用scratch创建base image 在Docker registry中有一个scratch，你可以pull拉取下来，

```
$ sudo docker pull scratch
```

甚至可以自己制作

```
$ tar cv --files-from /dev/null | docker import - scratch
```

Scratch镜像很赞，它简洁、小巧而且快速， 它没有bug、安全漏洞、延缓的代码或技术债务。这是因为它基本上是空的。除了Docker添加了点的metadata (译注：元数据为描述数据的数据)。总之它是非常小的一个Docker镜像。 为Scratch镜像创建内容，具体Dockerfile命令如下:

```
FROM scratch
ADD hello /
CMD ["/hello"]
```

3.下载官方提提供的OS的tar文件 到OPENVZ上下载基础包然后使用docker Iimport加载到本地镜像，这里以ubuntu14.04 为例，从openvz下载一个ubuntu14.04的模板：

```
wget http://download.openvz.org/template/precreated/ubuntu-14.04-x8
cat ubuntu-14.04-x86_64-minimal.tar.gz | docker import - ubuntu:bas
```

基本结构

Dockerfile 由一行行命令语句组成，并且支持以 # 开头的注释行。

一般的，Dockerfile 分为四部分：基础镜像信息、维护者信息、镜像操作指令和容器启动时执行指令。

例如

```
# This dockerfile uses the ubuntu image
# VERSION 2 - EDITION 1
# Author: docker_user
# Command format: Instruction [arguments / command] ..

# Base image to use, this must be set as the first line
FROM ubuntu

# Maintainer: docker_user <docker_user at email.com> (@docker_user)
MAINTAINER docker_user docker_user@email.com

# Commands to update the image
RUN echo "deb http://archive.ubuntu.com/ubuntu/ raring main univers
RUN apt-get update && apt-get install -y nginx
RUN echo "\ndaemon off;" >> /etc/nginx/nginx.conf

# Commands when creating a new container
CMD /usr/sbin/nginx
```

其中，一开始必须指明所基于的镜像名称，接下来推荐说明维护者信息。

后面则是镜像操作指令，例如 RUN 指令，RUN 指令将对镜像执行跟随的命令。每运行一条 RUN 指令，镜像添加新的一层，并提交。

最后是 CMD 指令，来指定运行容器时的操作命令。

下面是一个更复杂的例子

```
# Nginx
```

```
#
# VERSION               0.0.1

FROM      ubuntu
MAINTAINER Victor Vieux <victor@docker.com>

RUN apt-get update && apt-get install -y inotify-tools nginx apache

# Firefox over VNC
#
# VERSION               0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir /.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the tri
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example
#
# VERSION               0.1

FROM ubuntu
RUN echo foo > bar
# Will output something like ===> 907ad6c2736f

FROM ubuntu
RUN echo moo > oink
# Will output something like ===> 695d7793cbe4

# You'll now have two images, 907ad6c2736f with /bar, and 695d7793c
# /oink.
```

Dockerfile 操作建议

Docker可以读取一个Dockerfile文件来构建所需的镜像，这个文件里包含所有所需要的指令。Dockerfile文件用特有的格式来设置镜像信息，更多基础知识在 Dockerfile 参数详解 会详细展示。

本文包含Docker官方提供的一些践以及方法，我们强烈建议你去参照这些建议。

官方建议：

1. 一个Dockerfile文件尽量 `越简洁越好` ，这意味着 它可以被停止销毁，然后被最小化配置安装到另一个地方。
2. 在通常情况下，最好把 `Dockerfile文件放到一个空目录` ，然后，将构建镜像所需要动文件添加到该目录。为了提高构建性能，你也可以通过添加 `.dockerignore` 文件到该目录以排除文件和目录，该文件支持排斥的模式类似于.gitignore文件。
3. 为了 `减少镜像复杂度、依赖、文件大小和构建时间` ，应该尽量避免安装多余的不需要包，例如：你不需要在一个数据库镜像中添加一个文本编辑器。
4. 在绝大多数情况下， `每一个镜像只跑一个process` ，应用于多个容器中可以方边应用横向扩展和重复利用容器。如果该服务依赖于其他服务，请使用容器互联。
5. 你需要在Dockerfile的可读性和镜像层次最小化之间取得平衡，要有目的且非常谨慎的控制使用分层的数量
6. 尽可能缓解由字母数字排序的多行参数后的变化。这将帮助你避免包的重复，使列表更容易更新。这也使得PRs更容易审查。在一个空格前面加一个反斜杠能起到帮助。

   下面是来自buildpack-DEPS镜像中的例子：

   ```
   RUN apt-get update && apt-get install -y \
   bzr \
   cvs \
   git \
   mercurial \
   subversion
   ```

7. 构建缓存

在构建镜像时，进程将为 `Dockerfile内每一个指定的执行步骤构建一个镜像` 。由于执行每条指令都会对它缓存内现有镜像进行检查，所以 `镜像可以重复利用` ，而不是创建一个重复的镜像。如果你不想使用缓存，请在docker build 时使用 `--no-cach=true` 选项。

但是，如果你让Docker使用构建缓存找到匹配的镜像，你应该了解它什么时候需要，什么时候不需要，所以要遵循以下规则：

从缓存中启动一个基础镜像，执行下一条指令，比与上一层所有子镜像对比，`查看是否有与已经存在的镜像相同` ，如果不同，缓存将失效。

在大多数情况下，只需要简单地比较Dockerfile指令与其中一个子镜像就够了，但是，某些指令需要更多解释：

对于ADD和COPY指令，镜像中的文件 `每个的内容将全部检查` 。文件的某些信息是不检查的：最近更新时间和最后访问时间。在查找缓存期间，Docker会与已经存在的镜像文件进行校验，`如果文件被更改，例如内容或元数据，那么缓存也将失效` 。除此之外，ADD和COPY指令缓存将不会查看容器内文件来匹配缓存。例如，当执行一个 RUN apt-get -y 指令时，不会检查容器内文件更新来确定缓存是否存在。在这种情况下，将使用指令字符串本身来查找缓存匹配。

上文所提到动缓存失效，是指 `后续指令将会产生新的镜像文件，缓存将不会被使用` 。

镜像文件的大小

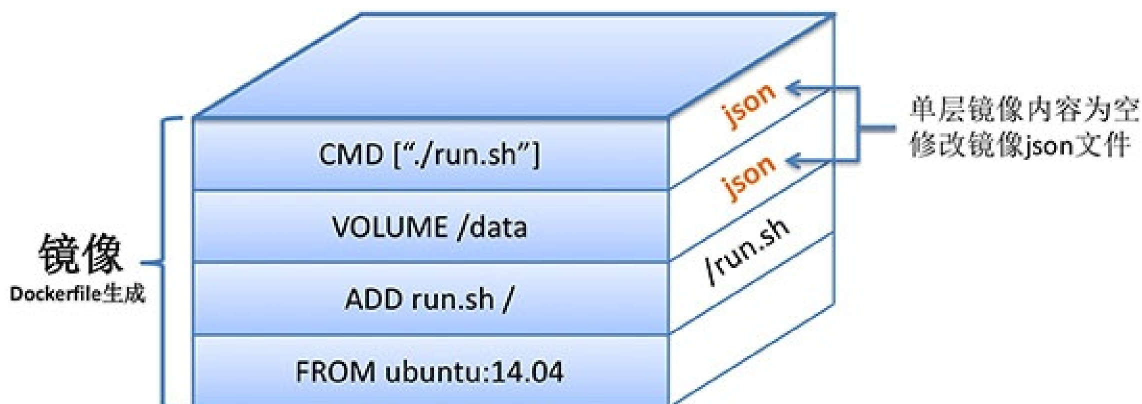
Dockerfile 与镜像

Dockerfile 由多条指令构成，随着深入研究 Dockerfile 与镜像的关系，很快大家就会发现，Dockerfile 中的 `每一条指令都会对应于 Docker 镜像中的一层` 。

以如下 Dockerfile 为例：

```
FROM ubuntu:14.04
ADD run.sh /
VOLUME /data
CMD ["./run.sh"]
```

通过 docker build 以上 Dockerfile 的时候，会在 ubuntu:14.04 镜像基础上，添加三层独立的镜像，依次对应于三条不同的命令。镜像示意图如下：



有了 Dockerfile 与镜像关系的初步认识之后，我们再进一步联系到每一层镜像的大小。

不得不说，在层级化管理的 Docker 镜像中，有不少层大小都为 0。那些镜像层大小不为 0 的情况，归根结底的原因是： `构建 Docker 镜像时，对当前的文件系统造成了修改更新` 。而修改更新的情况主要有两种：

1. ADD或 COPY 命令：ADD 或者 COPY 的作用是在 `docker build 构建镜像时向容器中添加内容` ，只要内容添加成功， `当前构建的那层镜像就是添加内容的大小` ，如以上命令 ADD run.sh /，新构建的那层镜像大小为文件 run.sh 的大小。

2. RUN 命令 `：RUN 命令的作用是在当前空的镜像层内运行一条命令` ，倘若运行的命令需要更新磁盘文件，那么所有的更新内容都在存储在当前镜像层中。举例说明： `RUN echo Hello world 命令不涉及文件系统内容的修改` ，故命令运行完之后当前镜像层的 `大小为 0` ； `RUN wget http://abc.com/def.tar` 命令会将压缩包下载至当前目录下，因此当前这一层镜像的大小为：对文件系统内容的增量修改部分，即 `def.tar 文件的大小` （在成功执行的情况下）。

联合文件系统

Dockerfile 中 `命令与镜像层一一对应` ，那么是否意味着 docker build 完毕之后，镜像的 `总大小是否等于每一层镜像的大小总和` 呢？答案是 `肯定的` 。依然以上图为例：如果 ubuntu:14.04 镜像的大小为 200 MB，而 run.sh 的大小为 5 MB，那么以上三层镜像从上到下，每层大小依次为 0、0 以及 5 MB，那么最终构建出的镜像大小的确为 0 + 0 + 5 + 200 = 205 MB。

虽然 `最终镜像的大小是每层镜像的累加` ，但是需要额外注意的是：Docker `镜像的大小并不等于容器中文件系统内容的大小` （不包括挂载文件，/proc、/sys 等虚拟文件）。个中缘由，就和联合文件系统有很大的关系了。

首先来看一下这个简单的 Dockerfile 例子（假如在 Dockerfile 当前目录下有一个 100 MB 的压缩文件 compressed.tar）：

```
FROM ubuntu:14.04
ADD compressed.tar /
RUN rm /compressed.tar
ADD compressed.tar /
```

1. FROM ubuntu:14.04：镜像 ubuntu:14.04 的 `大小为 200 MB` ；
2. ADD compressed.tar /： compressed.tar 文件为 100 MB，因此 `当前镜像层的大小为 100 MB` ，镜像 `总大小为 300 MB` ；
3. RUN rm /compressed.tar：删除文件 compressed.tar，此时的删 `除并不会删除下一层的 compressed.tar 文件` ，只会在当前层 `产生一个 compressed.tar 的删除标记` ，确保通过该层将看不到 compressed.tar，因此 `当前镜像层的大小也为 0` ，镜像 `总大小为 300 MB` ；
4. ADD compressed.tar /：compressed.tar 文件为 100 MB，因此 `当前镜像层的大小为 300 MB + 100 MB` ，镜像 `总大小为 400 MB` ；

分析完毕之后，我们发现镜像的总大小为 400 MB，但是如果 `运行该镜像` 的话，我们很快可以发现在容器根目录下执行 du -sh 之后，显示的数值并非 400 MB，而是 `300 MB 左右` 。主要的原因还是：联合文件系统的性质保 `证了两个拥有 compressed.tar 文件的镜像层，容器仅能看到一个` 。同时这也说明了一个现状，当用户基于一个非常大，甚至好几个 GB 的镜像运行容器时，在容器内部查看根目录大小，发现竟然只有 500 MB 不到，甚至更小。

分析至此，有一点大家需要非常注意： `镜像大小和容器大小有着本质的区别` 。

镜像共享关系

Docker 镜像说大不大，说小不小，但是一旦 `镜像的总数上来` 之后，岂不是对本地磁盘造成很大的存储压力？平均每个镜像 500 MB，岂不是 100 个镜像就需要准备 50 GB 的存储空间？

结果往往不是我们想象的那样，Docker 在 `镜像复用` 方面设计得非常出色，大大节省镜像占用的磁盘空间。Docker 镜像的复用主要体现在： `多个不同的 Docker 镜像可以共享相同的镜像层` 。

假设本地镜像存储中只有一个 ubuntu:14.04 的镜像，我们以两个 Dockerfile 来说明镜像复用：

```
FROM ubuntu:14.04
RUN apt-get update
FROM ubuntu:14.04
ADD compressed.tar /
```

假设最终 docker build 构建出来的镜像名分别为 image1 和 image2，由于两个 Dockerfile 均 `基于 ubuntu:14.04` ，因此，image1 和 image2 这两个镜像均 `复用了镜像 ubuntu:14.04` 。 假设 `RUN apt-get update` 修改的文件系统内容为 20 MB，最终本地三个镜像的大小关系应该如下：

```
ubuntu:14.04： 200 MB
image1：200 MB（ubuntu:14.04 的大小）+ 20 MB = 220 MB
image2：200 MB（ubuntu:14.04 的大小）+ 100 MB = 300 MB
```

如果仅仅是单纯的累加三个镜像的大小，那结果应该是： `200 + 220 + 300 = 720 MB` ，但是由于镜像复用的存在，实际占用的磁盘空间大小是 `：200 + 20 + 100 + 320 MB` ，足足节省了 400 MB 的磁盘空间。在此，足以证明镜像复用的巨大好处。

Dockerfile 参数详解

指令的一般格式为 `INSTRUCTION arguments` ，指令包括 `FROM、MAINTAINER、RUN` 等。

FROM

```
格式为 FROM <image>或FROM <image>:<tag>。
```

第一条指令必须为 `FROM` 指令。并且，如果在同一个Dockerfile中创建多个镜像时，可以使用多个 `FROM` 指令（每个镜像一次）。

MAINTAINER

```
格式为 MAINTAINER <name>，指定维护者信息。
```

RUN

```
格式为 RUN <command> 或 RUN ["executable", "param1", "param2"]。
```

`前者将在 shell 终端中运行命令` ，即 /bin/sh -c ； `后者则使用 exec 执行` 。指定使用其它终端可以通过第二种方式实现，例如 `RUN ["/bin/bash", "-c", "echo hello"]。`

每条 `RUN` 指令将在当前镜像基础上执行指定命令，并提交为新的镜像。当命令较长时可以使用 \ 来换行。

CMD

支持三种格式

```
CMD ["executable","param1","param2"] 使用 exec 执行，推荐方式；
CMD command param1 param2 在 /bin/sh 中执行，提供给需要交互的应用；
CMD ["param1","param2"] 提供给 ENTRYPOINT 的默认参数；
```

指定启动容器时执行的命令，每个 Dockerfile 只能有一条 `CMD` 命令。如果指定了多条命令，只有最后一条会被执行。

如果用户启动容器时候指定了运行的命令，则会覆盖掉 `CMD` 指定的命令。

EXPOSE

格式为

```
EXPOSE <port> [<port>...]。
```

告诉 Docker 服务端容器暴露的端口号，供互联系统使用。在启动容器时需要通过 `-P` ，Docker 主机会自动分配一个端口转发到指定的端口。

ENV

格式为

```
ENV <key> <value>。
```

指定一个环境变量，会被后续 `RUN` 指令使用，并在容器运行时保持。

例如

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -
ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

ADD

格式为

```
ADD <src> <dest>。
```

该命令将复制指定的 到容器中的 。 其中 可以是Dockerfile所在目录的一个相对路径；也可以是一个 URL；还可以是一个 tar 文件（自动解压为目录）。

COPY

格式为

```
COPY <src> <dest>。
```

复制本地主机的 （为 Dockerfile 所在目录的相对路径）到容器中的 。

当使用本地目录为源目录时，推荐使用 COPY。

ENTRYPOINT

两种格式：

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2（shell中执行）。
```

配置容器启动后执行的命令，并且不可被 `docker run` 提供的参数覆盖。

每个 Dockerfile 中只能有一个 `ENTRYPOINT` ，当指定多个时，只有最后一个起效。

VOLUME

格式为

```
VOLUME ["/data"]。
```

创建一个可以从本地主机或其他容器挂载的挂载点，一般用来存放数据库和需要保持的数据等。

USER

格式为

```
USER daemon。
```

指定运行容器时的用户名或 UID，后续的 RUN 也会使用指定用户。

当服务不需要管理员权限时，可以通过该命令指定运行用户。并且可以在之前创建所需要的用户，例如：`RUN groupadd -r postgres && useradd -r -g postgres postgres` 。`要临时获取管理员权限可以使用 gosu，而不推荐 sudo` 。

WORKDIR

格式为

```
WORKDIR /path/to/workdir。
```

为后续的 `RUN、CMD、ENTRYPOINT` 指令配置工作目录。

可以使用多个 `WORKDIR` 指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。例如

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

则最终路径为 `/a/b/c` 。

ONBUILD

格式为

```
ONBUILD [INSTRUCTION]。
```

配置当所创建的镜像作为其它新创建镜像的基础镜像时，所执行的操作指令。

例如，Dockerfile 使用如下的内容创建了镜像 image-A。

```
[...]
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build --dir /app/src
[...]
```

如果基于 image-A 创建新的镜像时，新的Dockerfile中使用 FROM image-A指定基础镜像时，会自动执行 ONBUILD 指令内容，等价于在后面添加了两条指令。

```
FROM image-A
#Automatically run the following
ADD . /app/src
RUN /usr/local/bin/python-build --dir /app/src
```

使用 ONBUILD 指令的镜像，推荐在标签中注明，例如 `ruby:1.9-onbuild` 。

创建镜像

编写完成 Dockerfile 之后，可以通过 docker build 命令来创建镜像。

基本的格式为 docker build [选项] 路径，该命令将读取指定路径下（包括子目录）的 Dockerfile，并将该路径下所有内容发送给 Docker 服务端，由服务端来创建镜像。因此一般建议放置 Dockerfile 的目录为空目录。也可以通过 .dockerignore 文件（每一行添加一条匹配模式）来让 Docker 忽略路径下的目录和文件。

要指定镜像的标签信息，可以通过 -t 选项，例如

```
$ sudo docker build -t myrepo/myapp /tmp/test1/
```

docker built 详细用法请点这里

容器

容器是 Docker 又一核心概念。

简单的说，容器是独立运行的一个或一组应用，以及它们的运行态环境。对应的，虚拟机可以理解为模拟运行的一整套操作系统（提供了运行态环境和其他系统环境）和跑在上面的应用。本章节着重介绍了容器的基础使用方法，以及如何管理容器的数据、如何管理容器的网络，相信你读完本章节将会有一个大致的了解。

容器使用入门

使用info命令检查docker安装程序是否正常运行：

```
# 检查是否安装好docker
$ docker info
```

如果提示如下信息: command not found 或者 类似于/var/lib/docker/repositories: permission denied 可能安装有问题或者尝试在前面加上sudo。

此外，基于docker系统配置，命令行前面应该加上sudo，来确保正常执行命令，并且此时系统会为Administrar创建一个名叫docker的Unix 用户组来让其他用户加入该组。

下载一个已经制作好的镜像

```
$ docker pull ubuntu
```

这条命令会从Docker Hub上搜索ubuntu镜像，然后下载到本里镜像缓存中去。

Note:When the image is successfully downloaded, you see a 12 character hash 539c0211cd76: Download complete which is the short form of the image ID. These short image IDs are the first 12 characters of the full image ID - which can be found using docker inspect or docker images --no-trunc=true.

创建一个带有交互窗口的**container**

```
$ docker run -i -t ubuntu /bin/bash
```

—i参数表示启动了一个可以交互的容器，—t参数表示创建了一个附带标准输入和输出的pseudo－TTY窗口

如果想要退出tty窗口，使用 Ctrl-p + Ctrl-q指令，容器将会退出并且会持续保持一个停止的状态。如果想查看所有状态的容器，可以使用docker ps －a 指令。

绑定容器到另外一个主机/端口或者一个Unix Socket

Warning: Changing the default docker daemon binding to a TCP port or Unix docker user group will increase your security risks by allowing non-root users to gain root access on the host. Make sure you control access to docker. If you are binding to a TCP port, anyone with access to that port has full Docker access; so it is not advisable on an open network.

使用 -H 能够让Docker deamon监听特殊的IP和端口。默认情况下，它将监听unix:///var/run/docker.sock以仅允许root进行本地连接。你可以将它设置为0.0.0.0:2375或者是指定的主机IP以供所有人连接，但是并不建议这么做，因为这将使某些无聊的人也获得deamon运行主机root的访问权限。

类似的，Docker客户端也可以使用 —H 连接一个指定端口

-H 使用以下格式来分配主机和端口：

tcp://[host][:port][path] or unix://path

例如:

- tcp://host:2375 -> TCP connection on host:2375
- tcp://host:2375/path -> TCP connection on host:2375 and prepend path to all requests
- unix://path/to/socket -> Unix socket located at path/to/socket

当—H参数为空时，将被默认认为没有—H参数传进来

-H also accepts short form for TCP bindings:

host[:port] or :port

Docker以daemon模式运行:

```
$ sudo <path to>/docker daemon -H 0.0.0.0:5555 &
```

下载一个ubuntu镜像:

```
$ docker -H :5555 pull ubuntu
```

如果你想同时监听TCP和Unix Socket 你可以添加多个 -H

```
# Run docker in daemon mode
$ sudo <path to>/docker daemon -H tcp://127.0.0.1:2375 -H unix:///v
# Download an ubuntu image, use default Unix socket
$ docker pull ubuntu
# OR use the TCP port
$ docker -H tcp://127.0.0.1:2375 pull ubuntu
```

起一个持续工作的进程

```
# Start a very useful long-running process
$ JOB=$(docker run -d ubuntu /bin/sh -c "while true; do echo Hello

# Collect the output of the job so far
$ docker logs $JOB

# Kill the job
$ docker kill $JOB
```

监听容器

```
$ docker ps # Lists only running containers
$ docker ps -a # Lists all containers
```

管理容器

```
# Start a new container
$ JOB=$(docker run -d ubuntu /bin/sh -c "while true; do echo Hello

# Stop the container
$ docker stop $JOB

# Start the container
$ docker start $JOB

# Restart the container
$ docker restart $JOB

# SIGKILL a container
$ docker kill $JOB

# Remove a container
$ docker stop $JOB # Container must be stopped to remove it
$ docker rm $JOB
```

在一个TCP端口上绑定一个服务

```
# Bind port 4444 of this container, and tell netcat to listen on it
$ JOB=$(docker run -d -p 4444 ubuntu:12.10 /bin/nc -l 4444)

# Which public port is NATed to my container?
$ PORT=$(docker port $JOB 4444 | awk -F: '{ print $2 }')

# Connect to the public port
$ echo hello world | nc 127.0.0.1 $PORT

# Verify that the network connection worked
$ echo "Daemon received: $(docker logs $JOB)"
```

提交（保存）一个容器的状态到一个镜像文件中

当提交（commit）容器时，docker仅保存源镜像与当前镜像的差异，如果想列出镜像，请使用docker images指令

```
# Commit your container to a new named image
$ docker commit <container> <some_name>

# List your images
$ docker images
```

管理容器工作

概览

我们用docker run指令来运行一个容器:

- 交互容器跑在前端.
- 守护进程跑在后台.

一些常用管理容器的命令:

- docker ps - 列出容器.
- docker logs - 输出容器日志.
- docker stop - 停止运行容器.

Docker客户端非常简单，你只需要需要输入一些带有一系列参数的指令就可以：

```
# Usage:  [sudo] docker [command] [flags] [arguments] ..
# Example:
$ docker run -i -t ubuntu /bin/bash
```

我们以docker version为例查看当前安装的docker客户端以及deamon进程信息

```
$ docker version
```

这条指令不仅提供啦docker客户端以及deamon进程的版本信息，同时也提供了所用go语言的信息。

```
Client:
  Version:      1.8.1
  API version:  1.20
  Go version:   go1.4.2
  Git commit:   d12ea79
  Built:        Thu Aug 13 02:35:49 UTC 2015
  OS/Arch:      linux/amd64

Server:
  Version:      1.8.1
  API version:  1.20
  Go version:   go1.4.2
  Git commit:   d12ea79
  Built:        Thu Aug 13 02:35:49 UTC 2015
  OS/Arch:      linux/amd64
```

获取docker命令行帮助

如果你想展示一些帮助信息，可以使用：

```
$ docker --help
```

如果你想了解具体参数的使用方法，可以参照如下使用：

```
$ docker attach --help

Usage: docker attach [OPTIONS] CONTAINER

Attach to a running container

--help=false        Print usage
--no-stdin=false    Do not attach stdin
--sig-proxy=true    Proxy all received signals to the process
```

在docker中跑一个web应用程序

现在你已经掌握了一些基础了，来深入一下吧。之前跑的一些应用没什么实际用途，让们来跑一个web应用试试吧。

这个web应用包含里 python 应用：

```
$ docker run -d -P training/webapp python app.py
```

以上指令中包含两个参数：

```
—d 让容器在后台运行
—P 分配一个主机端口到容器端口的映射以供外部访问
```

training/webapp是一个构建好的镜像，里面包含了一个简单的Python Flask web应用程序。

查看web应用程序状态

我们利用docker ps 来查看容器状态

```
$ docker ps -l
CONTAINER ID  IMAGE                     COMMAND        CREATED
bc533791f3f5  training/webapp:latest   python app.py 5 seconds ago
```

其中指令包含了一个—l参数，这将展示最近一次创建的容器的状态。

```
Note: 默认的 docker ps 不加参数将只会展现正在运行对容器，使用docker ps —a
```

在POTS栏中，我们可以查看到端口映射。

```
PORTS
0.0.0.0:49155->5000/tcp
```

当我们使用—P参数时，docker将会给主机指定容器暴露的所有端口。

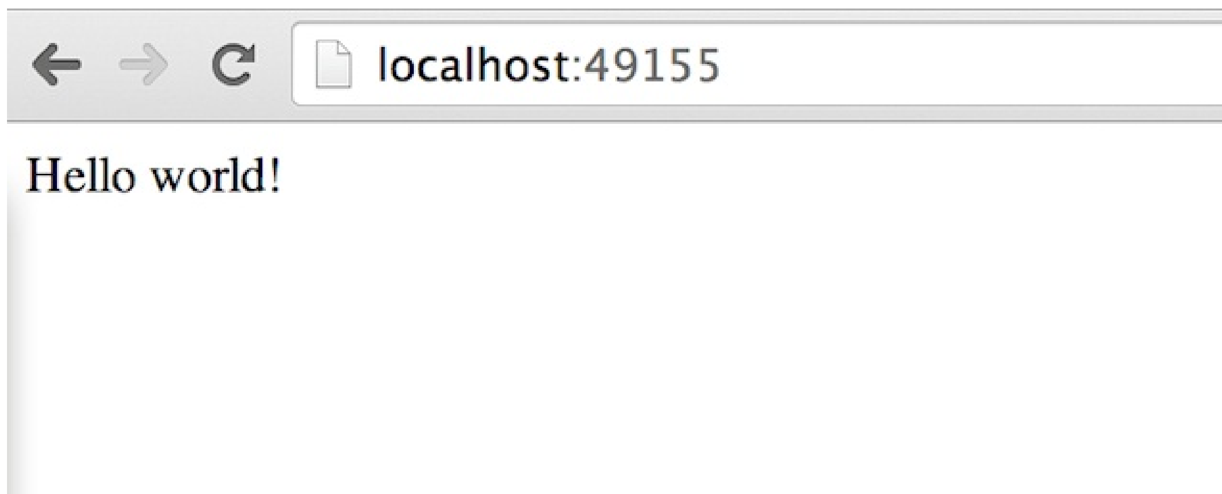
在这个例子中，容器暴露里5000端口，主机随机分配了一个49155端口与之映射。

网络端口绑定在Docker是可配置性非常高的。—P参数是随机指定一个32768～61000的主机端口与容器绑定，而－p则是指定一个端口与容器绑定：

```
$ docker run -d -p 80:5000 training/webapp python app.py
```

这条指令将绑定主机的80端口与容器的5000端口。

我们来看看49155端口上的web应用吧！



快捷查看网络端口

用docker ps命令可以查看端口映射，此外，docker海提供了另一种便捷的方式，用docker port命令

```
$ docker port nostalgic_morse 5000
0.0.0.0:49155
```

查看web应用日志

使用docker logs 可以查案容器内部的日志记录这样可以让我们清楚地看到容器运行的状态和历史纪录：

```
$ docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 40
```

这里加了一个－f参数，可以查看容器标准输出。这里展现了该web应用在5000端口上的日志信息。

查看web应用容器中的进程

根据容器的日志信息，我们可以利用 docker top 指令查看容器内的进程信息

```
$ docker top nostalgic_morse
PID                 USER                COMMAND
854                 root                python app.py
```

这里我们在容器里看到了刚才我们运行的 python app.py命令。

审查（inspect）web应用容器

利用docker inspect 查看一个容器的详细信息（包括运行状态）或者镜像的配置信息，展现出来的是一个json文件。

```
$ docker inspect nostalgic_morse
Let’s see a sample of that JSON output.

[{
    "ID": "bc533791f3f500b280a9626688bc79e342e3ea0d528efe3a86a51ecb
    "Created": "2014-05-26T05:52:40.808952951Z",
    "Path": "python",
    "Args": [
       "app.py"
    ],
    "Config": {
       "Hostname": "bc533791f3f5",
       "Domainname": "",
       "User": "",
    . . .
```

我们可以使用-f参数里筛选出需要的信息

```
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' nostalgic_mo
172.17.0.5
```

停止web应用容器

现在我们尝试停止我们刚才启用的容器，容器名称: nostalgic_morse.

```
$ docker stop nostalgic_morse
nostalgic_morse
```

我们可以使用docker ps 来查看是否成功

```
$ docker ps -l
```

重启web应用容器

重启一下试试：

```
$ docker start nostalgic_morse
nostalgic_morse
```

用docker ps －l命令或者打开浏览器产看变化

删除web应用容器

我们可以利用docker rm来删除一个容器，但有时候会出现以下错误：

```
$ docker rm nostalgic_morse
Error: Impossible to remove a running container, please stop it fi
2014/05/24 08:12:56 Error: failed to remove one or more containers
```

怎么回事? 原来删除的是一个正在运行的容器. 所以我们要做的是stop it&remove it

```
$ docker stop nostalgic_morse
nostalgic_morse
$ docker rm nostalgic_morse
nostalgic_morse
```

官方建议

```
Note: Always remember that removing a container is final!
```

管理容器数据

到目前为止，我们已经介绍了一些基本的docker概念，如何管理docker 镜像，以及了解网络和容器之间的联系。

在这一节中，我们将介绍如何管理容器数据。

docker管理数据的两种主要方式。

数据卷，以及数据卷容器。

数据卷是在一个或多个容器，它绕过Union File System的一个专门指定的目录。数据卷为持续共享数据提供了一些有用的功能：

- 在创建容器时，卷被初始化。如果容器的基础映像包含指定的数据装入点，现有的数据复制到在卷初始化新卷。
- 数据卷可以共享和容器之间重复使用。
- 改变数据卷将立刻生效（在所有挂载该容器中）
- 改变数据卷数据不会影响到容器。
- 即使容器本身被删除。但是数据卷依然存在。
- 数据卷的目的是持久化数据，独立于容器的生命周期。Docker因此不会自动删除卷，当你删除一个容器，也不会"垃圾回收"直到没有容器再使用。

添加一个数据卷

你可以在docker run时加上－v参数来添加一个数据卷，－v参数也可以使用多次，以挂载多个数据卷。

```
$ docker run -d -P --name web -v /webapp training/webapp python app
```

这条命令将在容器中的／webapp文件夹创建一个数据卷存储数据。

你也可以在构建镜像时在Dockerfile里面定义。

数据卷默认的权限时读写，你也可以定义成只读。

```
$ docker run -d -P --name web -v /opt/webapp:ro training/webapp pyt
```

查找数据卷路径用docker inspect命令定位

```
$ docker inspect web
```

我们可以看到保存在主机上数据卷的路径：

```
...
Mounts": [
    {
        "Name": "fac362...80535",
        "Source": "/var/lib/docker/volumes/fac362...80535/_data",
        "Destination": "/webapp",
        "Driver": "local",
        "Mode": "",
        "RW": true
    }
]
```

... 并切看到权限RW是ture

挂载一个主机目录作为数据卷

挂载主机目录为数据卷，必须参照 -v hostPATH:containerPATH 这种格式 路径必须为绝对路径，以保证容器的 可移植性。

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp training/w
```

上面的命令加载主机的 /src/webapp 目录到容器的 /opt/webapp 目录

docker数据卷的权限是读写，你也可以指定只读：

```
$ docker run -d -P --name web -v /src/webapp:/opt/webapp:ro trainin
```

挂载一个数据文件作为数据卷

-v 标记也可以从主机挂载单个文件到容器中

```
$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu
```

这样就可以记录在容器输入过的命令了。

如果直接挂载一个文件，很多文件编辑工具，包括 vi 或者 sed --in-place，可能会造成文件 inode 的改变，从 Docker 1.1 .0起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

创建和挂载数据卷容器

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。

数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。

首先，创建一个命名的数据卷容器 dbdata：

```
$ sudo docker run -d -v /dbdata --name dbdata training/postgres ech
```

然后，在其他容器中使用 --volumes-from 来挂载 dbdata 容器中的数据卷。

```
$ sudo docker run -d --volumes-from dbdata --name db1 training/post
$ sudo docker run -d --volumes-from dbdata --name db2 training/post
```

还可以使用多个 --volumes-from 参数来从多个容器挂载多个数据卷。 也可以从其他已经挂载了数据卷的容器来挂载数据卷。

```
$ sudo docker run -d --name db3 --volumes-from db1 training/postgre
```

*注意：使用 --volumes-from 参数所挂载数据卷的容器自己并不需要保持在运行状态。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。

备份、存储、移动数据卷

另一个非常有用大功能是利用数据卷容器进行备份、存储以及迁移操作。

备份

```
$ docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf
```

然后新创建一个新的容器

```
$ docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后解压数据卷挂载到容器

```
$ docker run --volumes-from dbdata2 -v $(pwd):/backup ubuntu cd /db
```

容器间的通信

容器的通信相当重要，这里讲解了一通信方式。

容器与宿主机器采用端口映射的方式通信

之前的例子

```
$ docker run -d -P training/webapp python app.py
```

我们可以看到端口映射状态：

```
$ docker ps nostalgic_morse
CONTAINER ID  IMAGE                   COMMAND       CREATED
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago
```

我们也可以指定端口映射：

```
$ docker run -d -p 80:5000 training/webapp python app.py
```

－p参数还有其他指定方法：

```
ip:hostPort:containerPort  映射指定IP的指定端口

ip::containerPort          映射指定IP任意端口

hostPort:containerPort。     映射所有主机IP的指定端口
```

采用link方式通信

名字的重要性 为了方便建立连接，通常需要为容器起一个name， 如果不起， 你会发现系统会自动分配一个名字。

此外，起一个自定义的名字让你很容易地记住它并切可以方便的管理

```
$ docker run -d -P --name web training/webapp python app.py
```

查看容器

```
$ docker ps -l
CONTAINER ID  IMAGE                    COMMAND        CREATED       S
aed84ee21bde  training/webapp:latest python app.py  12 hours ago  U
```

你也可以使用docker inspect 命令查看容器名称。

```
Note: Container names have to be unique. That means you can only ca
```

Links 允许容器发现另一个容器，并在期间建立一个安全的通道以便交换数据。 使用－－link参数来创建一个连接。

```
$ docker run -d --name db training/postgres
```

创建一个web连接到db数据库容器.

```
$ docker run -d -P --name web --link db:db training/webapp python a
```

参数格式如下:

```
--link <name or id>:alias
```

alias代表你 为这个链接起的一个别名。

```
--link <name or id>
```

该参数将匹配容器name 并建立连接

```
$ docker run -d -P --name web --link db training/webapp python app.
```

使用docker inspect 定位:

```
$ docker inspect -f "{{ .HostConfig.Links }}" web
[/db:/web/db]
```

你可以看到web容器已经与db容器连接

连接容器docker究竟做了什么？目前已经知道，一个创建在源容器与目标容器间的连接允许源容器提供信息给目标容器。在上述例子中，目标容器web可以获取源容器db提供的信息。为了实现这项功能，docker在这两个容器之间创建了一个稳定的通道，并且不会暴露任何端口，你不需要在创建容器时加上－p或－P参数。这就是link方式的最大好处。

docker通过以下两种方式完成此项工作

1、环境变量

当link容器时，docker会创建许多环境变量. Docker会自动创建环境变量到目标容器中去. 它也将通过docker暴露源容器的所有环境变量. 这些变量来自:

Dockerfile中ENV命令定义的环境变量

在启动源容器中使用 －e 、--env以及 --env－file参数附加的环境变量。

这些环境变量使程序从相关的目标容器中发现源容器。

```
Warning: It is important to understand that all environment variabl
```

Docker为每一个在--link参数中的容器设置了一个 _NAME 环境变量 。例如，一个web容器通过--link db：webdb连 接db容器，将会在web容器中创建一个WEBDB_NAME=/web/webdb环境变量

Docker为源容器暴露的端口限定了一组环境变量，每一个环境变量具有唯一前缀形式：

```
<name>_PORT_<port>_<protocol>
```

前缀的构成:

- 是--link :后面的参数(例如, webdb)
- 就是暴露的端口号
- TCP／UDP

Docker 利用这前缀格式定义了三个不同的环境变量:

prefix_ADDR 变量包含了来自URL的IP地址, for example

```
WEBDB_PORT_5432_TCP_ADDR=172.17.0.82.
```

prefix_PORT 变量仅包含了URL的端口号, for example

```
WEBDB_PORT_5432_TCP_PORT=5432.
```

prefix_PROTO 参数包含URL的传输协议, for example

```
WEBDB_PORT_5432_TCP_PROTO=tcp.
```

如果容器暴露多个端口，Docker将会为每个端口创建三个环境变量。算术题：如果容器暴露4个端口，将会创建多少个环境变量？答对了，是12个哦！每个端口三个环境变量。

另外，Docker也要创建一个叫 _PORT 的环境变量。这个变量包含源容器URL首次暴露的IP和端口。该端口的“首次”定义为最低级数字的端口。例如，思考 WEBDB_PORT=tcp://172.17.0.82:5432 变量，如果该端口同时用语TCP和UDP，则TCP将会被指定。（原文：*Additionally, Docker creates an environment variable called _PORT. This variable contains the URL of the source container’s first exposed port. The ‘first’ port is defined as the exposed port with the lowest number. For example, consider the WEBDB_PORT=tcp://172.17.0.82:5432 variable. If that port is used for both tcp and udp, then the tcp one is specified.*）

最后，Docker会把源容器中的环境变量暴露给目标容器作为环境变量。并且Docker会在目标容器为每个变量创建一个*ENV*变量。这个变量的值被设置为启动源容器Docker所用到的值。（原文：*Finally, Docker also exposes each Docker originated environment variable from the source container as an environment variable in the target. For each variable Docker creates an ENV variable in the target container. The variable's value is set to the value Docker used when it started the source container.*）

回到之前的例子 database ,你可以使用env命令列出具体的容器环境变量：

```
$ docker run --rm --name web2 --link db:db training/webapp env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5432_TCP=tcp://172.17.0.5:5432
DB_PORT_5432_TCP_PROTO=tcp
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_ADDR=172.17.0.5
. . .
```

你可以看到Docker利用许多有关源容器的信息创建了一些列的环境变量。每一环境变量都会带有指点定义的别名，DB前缀。如果别名是*db1*，那么变量前缀也会变成*DB1*。利用这线环境变量来配置应用用来在db容器上连接数据库。这样的连接方式稳定安全私有化。只有已获得连接的web容器才会有对db容器的访问权限。

关于环境变量的一些注意事项：

与修改/etc/hosts文件不同，在环境变量中存储的IP地址信息不回随着容器的重启而更新，建议利用hosts文件来解决连接容器的IP地址问题。

这些环境变量只是为容器的第一个process设置，某些deamon后台服务，例如sshd，只有当产生连接需求时才会设置。（原文：These environment variables are only set for the first process in the container. Some daemons, such as sshd, will scrub them when spawning shells for connection.）

2、更新 /etc/hosts 文件

处理环境变量, docker在源文件中追加了host信息，这里向web容器追加:

```
$ docker run -t -i --rm --link db:webdb training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.5  webdb 6e5cdeb2d300 db
```

我们可以在容器中使用ping命令测试链接：

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping webdb
PING webdb (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

如果你重启源容器，连接依然存在：

```
$ docker restart db
db
$ docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.9  db
```

Docker 的基本指令及用法详解

Docker官方为了让用户快速了解Docker，提供了一个交互式教程，旨在帮助用户掌握Docker命令行的使用方法。但是由于Docker技术的快速发展，此交互式教程已经无法满足Docker用户的实际使用需求，所以让我们一起开始一次真正的命令行学习之旅。首先，Docker的命令清单可以通过运行 docker，或者 docker help 命令得到:

```
$ sudo docker --help
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.

Options:
--add-registry=[]                    Registry to query before a pub
  --api-cors-header=                   Set CORS headers in the remo
  -b, --bridge=                        Attach containers to a netwo
  --bip=                               Specify network bridge IP
  --block-registry=[]                  Don't contact given registry
  --confirm-def-push=true              Confirm a push to default re
  -D, --debug=false                    Enable debug mode
  -d, --daemon=false                   Enable daemon mode
  --default-gateway=                   Container default gateway IP
  --default-gateway-v6=                Container default gateway IP
  --default-ulimit=[]                  Set default ulimits for cont
  --dns=[]                             DNS server to use
  --dns-search=[]                      DNS search domains to use
  -e, --exec-driver=native             Exec driver to use
  --exec-opt=[]                        Set exec driver options
  --exec-root=/var/run/docker          Root of the Docker execdrive
  --fixed-cidr=                        IPv4 subnet for fixed IPs
  --fixed-cidr-v6=                     IPv6 subnet for fixed IPs
  -G, --group=docker                   Group for the unix socket
  -g, --graph=/var/lib/docker          Root of the Docker runtime
  -H, --host=[]                        Daemon socket(s) to connect
  -h, --help=false                     Print usage
  --icc=true                           Enable inter-container commu
  --insecure-registry=[]               Enable insecure registry com
```

```
  --ip=0.0.0.0                           Default IP when binding cont
  --ip-forward=true                      Enable net.ipv4.ip_forward
  --ip-masq=true                         Enable IP masquerading
  --iptables=true                        Enable addition of iptables
  --ipv6=false                           Enable IPv6 networking
  -l, --log-level=info                   Set the logging level
  --label=[]                             Set key=value labels to the
  --log-driver=json-file                 Default driver for container
  --log-opt=map[]                        Set log driver options
  --mtu=0                                Set the containers network M
  -p, --pidfile=/var/run/docker.pid      Path to use for daemon PID f
  --registry-mirror=[]                   Preferred Docker registry mi
  -s, --storage-driver=                  Storage driver to use
  --selinux-enabled=false                Enable selinux support
  --storage-opt=[]                       Set storage driver options
  --tls=false                            Use TLS; implied by --tlsver
  --tlscacert=~/.docker/ca.pem           Trust certs signed only by t
  --tlscert=~/.docker/cert.pem           Path to TLS certificate file
  --tlskey=~/.docker/key.pem             Path to TLS key file
  --tlsverify=false                      Use TLS and verify the remot
  --userland-proxy=true                  Use userland proxy for loopb
  -v, --version=false                    Print version information an

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders from a container's filesystem to t
    create    Create a new container
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    exec      Run a command in a running container
    export    Stream the contents of a container as a tar archive
    history   Show the history of an image
    images    List images
    import    Create a new filesystem image from the contents of a
    info      Display system-wide information
    inspect   Return low-level information on a container or image
    kill      Kill a running container
    load      Load an image from a tar archive
```

```
    login      Register or log in to a Docker registry server
    logout     Log out from a Docker registry server
    logs       Fetch the logs of a container
    pause      Pause all processes within a container
    port       Lookup the public-facing port that is NAT-ed to PRIVA
    ps         List containers
    pull       Pull an image or a repository from a Docker registry
    push       Push an image or a repository to a Docker registry se
    rename     Rename an existing container
    restart    Restart a running container
    rm         Remove one or more containers
    rmi        Remove one or more images
    run        Run a command in a new container
    save       Save an image to a tar archive
    search     Search for an image on the Docker Hub
    start      Start a stopped container
    stats      Display a stream of a containers' resource usage stat
    stop       Stop a running container
    tag        Tag an image into a repository
    top        Lookup the running processes of a container
    unpause    Unpause a paused container
    version    Show the Docker version information
    wait       Block until a container stops, then print its exit co

Run 'docker COMMAND --help' for more information on a command.
```

在Docker容器技术不断演化的过程中，Docker的子命令已经达到39个之多，其中核心子命令(例如：run)还会有复杂的参数配置。笔者通过结合功能和应用场景方面的考虑，把命令行划分为4个部分，方便我们快速概览Docker命令行的组成结构：

功能划分	命令
环境信息相关	info version
系统运维相关	attach build commit cp diff export images import save/load inspect kill port pause/unpause ps rm rmi run start/stop/restart tag top
日志信息相关	events history logs
仓库服务相关	login pull push search

1.参数约定

单个字符的参数可以放在一起组合配置，例如

```
$ sudo docker run -t -i --name test centos sh
```

可以用这样的方式等同：

```
$ sudo docker run -ti --name test centos sh
```

2.Boolean

Boolean参数形式如： -d=false。注意，当你声明这个Boolean参数时，比如 docker run -d=true，它将直接把启动的Container挂起放在后台运行。

3.字符串和数字

参数如 --name="" 定义一个字符串，它仅能被定义一次。同类型的如-c=0 定义一个数字，它也只能被定义一次。

4.后台进程

Docker后台进程是一个常驻后台的系统进程，值得注意的是Docker使用同一个文件来支持客户端和后台进程，其中角色切换通过-d来实现。这个后台进程是用来管理容器的:

参数	解释
--add-registry=[]	Registry to query before a public one
--api-cors-header=	Set CORS headers in the remote API
-b, --bridge=	挂载已经存在的网桥设备到 Docker 容器里。注意，使用 none 可以停用容器里的网络。
--bip=	使用 CIDR 地址来设定网络桥的 IP。注意，此参数和 -b 不能一起使用。
--block-registry=[]	Don't contact given registry
--confirm-def-push=true	Confirm a push to default registry
-D, --debug=false	开启Debug模式。例如：docker -d -D
-d, --daemon=false	开启Daemon模式。
--default-gateway=	Container default gateway IPv4 address

--default-gateway-v6=	Container default gateway IPv6 address
--default-ulimit=[]	Set default ulimits for containers
--dns=[]	强制容器使用DNS服务器。例如： docker -d --dns 8.8.8.8
--dns-search=[]	强制容器使用指定的DNS搜索域名。例如： docker -d --dns-search example.com
-e, --exec-driver=native	强制容器使用指定的运行时驱动。例如：docker -d -e lxc
--exec-opt=[]	Set exec driver options
--exec-root=/var/run/docker	Root of the Docker execdriver
--fixed-cidr=	IPv4 subnet for fixed IPs
--fixed-cidr-v6=	IPv6 subnet for fixed IPs
-G, --group=docker	在后台运行模式下，赋予指定的Group到相应的unix socket上。注意，当此参数 --group 赋予空字符串时，将去除组信息。
-g, --graph=/var/lib/docker	配置Docker运行时根目录
-H, --host=[]	Daemon socket(s) to connect to

-h, --help=false	在后台模式下指定socket绑定，可以绑定一个或多个 tcp://host:port, unix:///path/to/socket, fd://* 或 fd://socketfd。例如：
	$ docker -H tcp://0.0.0.0:2375 ps 或者
	$ export DOCKER_HOST="tcp://0.0.0.0:2375"
	$ docker ps

--icc=true	启用内联容器的通信。
--insecure-registry=[]	Enable insecure registry communication
--ip=0.0.0.0	容器绑定IP时使用的默认IP地址
--ip-forward=true	Enable net.ipv4.ip_forward
--ip-masq=true	Enable IP masquerading
--iptables=true	启动Docker容器自定义的iptable规则
--ipv6=false	Enable IPv6 networking
-l, --log-level=info	Set the logging level
--label=[]	Set key=value labels to the daemon
--log-driver=json-file	Default driver for container logs
--log-opt=map[]	Set log driver options
--mtu=0	设置容器网络的MTU值，如果没有这个参数，选用默认 route MTU，如果没有默认route，就设置成常量值 1500。
-p, --	后台进程PID文件路径。

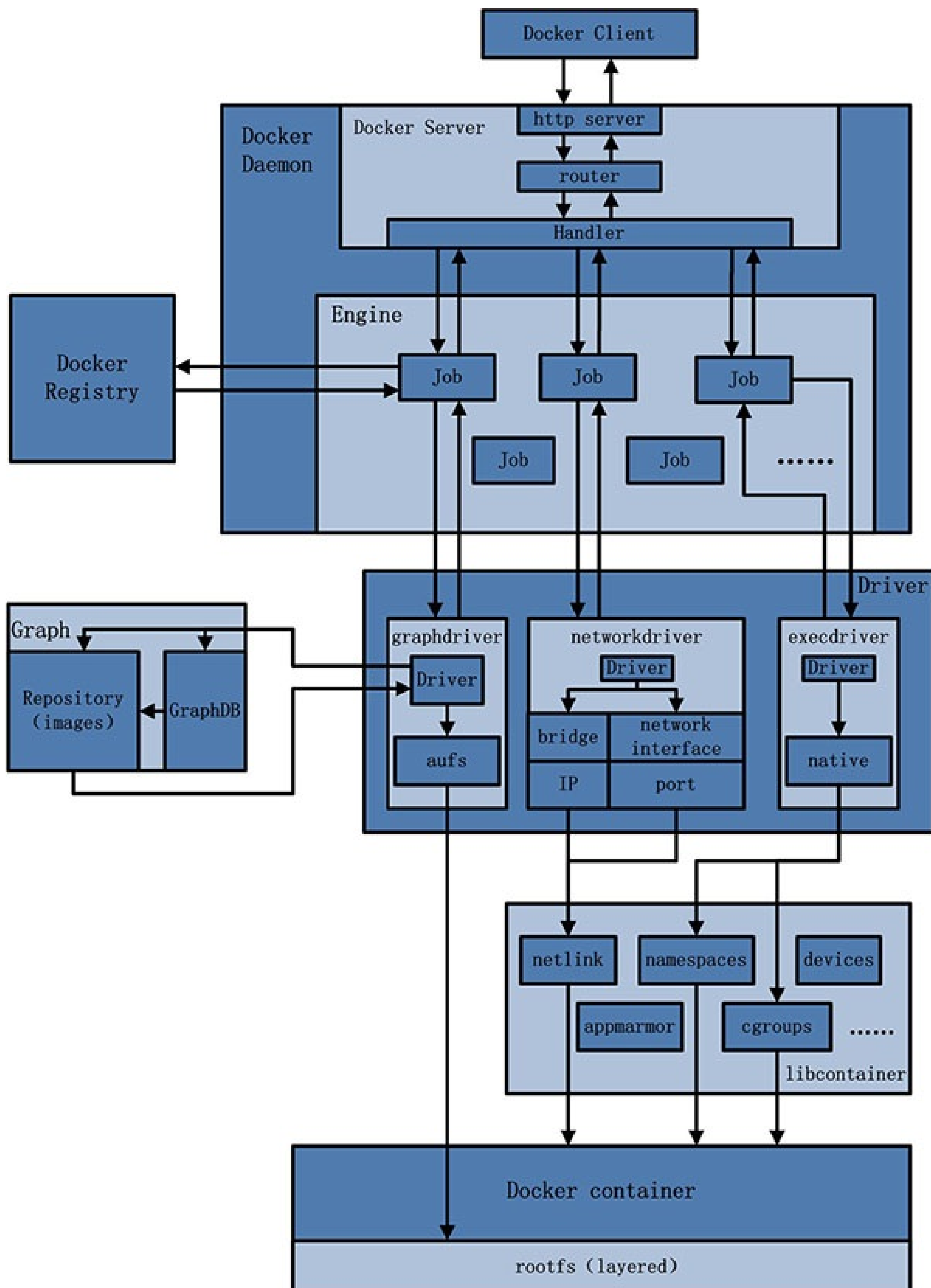
pidfile=/var/run/docker.pid	后台进程PID文件路径。
--registry-mirror=[]	Preferred Docker registry mirror
-s, --storage-driver=	强制容器运行时使用指定的存储驱动，例如,指定使用devicemapper, 可以这样： $ sudo docker -d -s devicemapper
--selinux-enabled=false	启用selinux支持
--storage-opt=[]	配置存储驱动的参数
--tls=false	启动TLS认证开关
--tlscacert=~/.docker/ca.pem	通过CA认证过的的certificate文件路径
--tlscert=~/.docker/cert.pem	TLS的certificate文件路径
--tlskey=~/.docker/key.pem	TLS的key文件路径
--tlsverify=false	使用TLS并做后台进程与客户端通讯的验证
--userland-proxy=true	Use userland proxy for loopback traffic
-v, --version=false	显示版本信息

注意，其中带有[] 的启动参数可以指定多次，例如:

```
$ sudo docker run -a stdin -a stdout -a stderr -i -t ubuntu /bin/ba
```

deamon＊

Docker对使用者来讲是一个 `C/S` 模式的架构，而Docker的后端是一个非常松耦合的架构，模块各司其职，并有机组合，支撑Docker的运行,如下图所示：



不难看出，用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。 而Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；而后Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。

Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。

而libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作。

当执行完运行容器的命令后，一个实际的Docker容器就处于运行状态，该容器拥有独立的文件系统，独立并且安全的运行环境等。

Docker Client

Docker Client是Docker架构中用户用来和Docker Daemon建立通信的客户端。用户使用的可执行文件为docker，通过docker命令行工具可以发起众多管理container的请求。

Docker Client可以通过以下三种方式和Docker Daemon建立通信：tcp://host:port，unix://path_to_socket和fd://socketfd。为了简单起见，本文一律使用第一种方式作为讲述两者通信的原型。与此同时，与Docker Daemon建立连接并传输请求的时候，Docker Client可以通过设置命令行flag参数的形式设置安全传输层协议(TLS)的有关参数，保证传输的安全性。

Docker Client发送容器管理请求后，由Docker Daemon接受并处理请求，当Docker Client接收到返回的请求相应并简单处理后，Docker Client一次完整的生命周期就结束了。当需要继续发送容器管理请求时，用户必须再次通过docker可执行文件创建Docker Client。

Docker Daemon

Docker Daemon是Docker架构中一个常驻在后台的系统进程，功能是：接受并处理Docker Client发送的请求。该守护进程在后台启动了一个Server，Server负责接受Docker Client发送的请求；接受请求后，Server通过路由与分发调度，找到相应的Handler来执行请求。

Docker Daemon启动所使用的可执行文件也为docker，与Docker Client启动所使用的可执行文件docker相同。在docker命令执行时，通过传入的参数来判别Docker Daemon与Docker Client。

deamon的参数选项：

```
Usage: docker daemon [OPTIONS]
A self-sufficient runtime for linux containers.
Options:
  --api-cors-header=""                    Set CORS headers in the re
  -b, --bridge=""                         Attach containers to a net
  --bip=""                                Specify network bridge IP
  -D, --debug=false                       Enable debug mode
  --default-gateway=""                    Container default gateway
  --default-gateway-v6=""                 Container default gateway
  --dns=[]                                DNS server to use
  --dns-search=[]                         DNS search domains to use
  --default-ulimit=[]                     Set default ulimit setting
  -e, --exec-driver="native"              Exec driver to use
  --exec-opt=[]                           Set exec driver options
  --exec-root="/var/run/docker"           Root of the Docker execdri
  --fixed-cidr=""                         IPv4 subnet for fixed IPs
  --fixed-cidr-v6=""                      IPv6 subnet for fixed IPs
  -G, --group="docker"                    Group for the unix socket
  -g, --graph="/var/lib/docker"           Root of the Docker runtime
  -H, --host=[]                           Daemon socket(s) to connec
  -h, --help=false                        Print usage
  --icc=true                              Enable inter-container com
  --insecure-registry=[]                  Enable insecure registry c
  --ip=0.0.0.0                            Default IP when binding co
  --ip-forward=true                       Enable net.ipv4.ip_forward
  --ip-masq=true                          Enable IP masquerading
  --iptables=true                         Enable addition of iptable
  --ipv6=false                            Enable IPv6 networking
  -l, --log-level="info"                  Set the logging level
  --label=[]                              Set key=value labels to th
  --log-driver="json-file"                Default driver for contain
  --log-opt=[]                            Log driver specific option
  --mtu=0                                 Set the containers network
  -p, --pidfile="/var/run/docker.pid"     Path to use for daemon PID
```

```
  --registry-mirror=[]                   Preferred Docker registry
  -s, --storage-driver=""                Storage driver to use
  --selinux-enabled=false                Enable selinux support
  --storage-opt=[]                       Set storage driver options
  --tls=false                            Use TLS; implied by --tlsv
  --tlscacert="~/.docker/ca.pem"         Trust certs signed only by
  --tlscert="~/.docker/cert.pem"         Path to TLS certificate fi
  --tlskey="~/.docker/key.pem"           Path to TLS key file
  --tlsverify=false                      Use TLS and verify the rem
  --userland-proxy=true                  Use userland proxy for loc
```

如果你想要运行守护态进程，你可以输入 `docker -d` （之前版本是 `docker deamon` ）。如果想加入Debug模式，输入 `docker -d -D` 即可。

- Deamon socket 选项

Docker deamon 通过三种不同的socket方式监听 `docker remote API` 请求，分别是： `unix、tcp、以及fd。`

默认情况下，通过创建在 `/var/run/docer.sock` 文件内的 `unix domain socket` （或者 IPC socket）来接收root或者docker用户组的请求。如果你想远程通信你需要打开tcpSocket。

要注意的是，默认的方式提供了一个未加密未验证直接连接deamon。应该使用内置的HTTPS加密的socket或者在前面使用一个安全的web代理。使用-H `tcp://0.0.0.0:2375` 来监听所有ip地址接口的2375端口，或者指定一个主机IP监听 `-H 192.168.2.160:2375` 。通常情况下2375端口是 未加密的，而2376用于与deamon通信的加密端口。

```
注意：如果你使用HTTPS加密socket ，目前支持TLS1.0或更高级的协议，不支持Proto
```

在Systemd基础的系统中，使用 `docker -d -H fd://` ,通过Systemd socket activation与deamon通信。对于大多数设置，使用fd://将很好的运作，你也可以指定单个socket： `docker -d -H fd://3` 。如果没有找到指定的激活的文件，Docker 将会退出进程。

1. **Server端**

   -H参数可以多次指定监听不同的端口：

例如指定监听主机默认的unix socket以及指定的IP地址：

```
$ sudo docker -d -H unix:///var/run/docker.sock  -H tcp://192.
```

2. **Client端**

为客户端设置-H参数，将使客户端监听 `DOCKER_HOST` 环境变量指定的参数：

```
$ docker -H tcp://0.0.0.0:2375 ps
```

或者

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ docker ps
```

设置 `DOCKER_TLS_VERIFY` 环境变量相当于设置 `--tlsverify` 参数：

```
$ docker --tlsverify ps
```

或者

```
$ export DOCKER_TLS_VERIFY=1
$ docker ps
```

以上设置是等效的

Docker客户端会遵守 `HTTP_PROXY,HTTPS_PROXY以及NO_PROXY` 这三个环境变量运行。其中 `HTTPS_PROXY` 优先权大于 `HTTP_PROXY`

- storage-driver 选项

Docker deamon 支持许多不同的镜像层存储驱动：aufs、devicemapper、btrfs、zfs以及overlay。

1. aufs是最老的，但是由于它是基于linux 内核patch-set,不太可能被合并到主内核中。这也会导致一些严重的系统崩溃。但是，aufs也是唯一允许容器共享可执行文件以及共享类库内存的存储驱动，所以对于那些需要运行数以千计运行相同程序或类库的容器会非常有用。

2. devicemapper使用自动精简配置以及Copy on Write(COW)快照。对于每一个graph位置通常是在/var/lib/docker/devicemapper中，通常被分为两块设备，一块给数据，一块给metadata。默认的，这些块设备是通过使用自动创建的零散文件回送挂载来自动创建的。Refer to Storage driver options below for a way how to customize this setup.~jpetazzo/Resizing Docker containers with the Device Mapper plugin article explains how to tune your existing setup without the use of options.

3. Btrfs 对于docker build构建镜像时会非常快，但是和devicemapper一样不会共享可执行文件以及类库的内存。使用方法：

   ```
   docker -d -s btrfs -g /mnt/btrfs_partition
   ```

4. Zfs 没有btrfs那么快，但是对相对较长记录有更稳定地支持。由于克隆之间的单一副本ARC共享块将被一次缓存，使用方法：

   ```
   docker -d -s zfs
   ```

   Use docker daemon -s zfs. To select a different zfs filesystem set zfs.fsname option as described in Storage driver options.

5. Overlay 是一个非常快的联合文件系统，它现在被并入了3.18.0的Linux内核中，使用方法：

   ```
   docker -d -s overlay
   ```

- storage-opt选项

  `dm.thinpooldev` ,指定块存储设备所使用的thin pool。

  ```
  docker -d --storage-opt dm.thinpooldev=/dev/mapper/thin-pool
  ```

`dm.basesize` 指定基础存储大小，同时限制镜像以及容器。默认值时100G。 修改此值需要执行以下操作才生效：

```
$ sudo service docker stop
$ sudo rm -rf /var/lib/docker
$ sudo service docker start
```

使用方法：

```
$ docker -d --storage-opt dm.basesize=20G
```

`dm.loopdatasize` 这个选项配置devicemapper looback，这不应该在生产中使用。默认值是100G，用于设定thin pool为数据产生的回送的零散文件存储大小，通常不会占用那么多空间。

使用方法：

```
$ docker -d --storage-opt dm.loopdatasize=200G
```

`dm.loopmetadatasize` 与上面类似，只是设定元数据存储大小。

使用方法

```
$ docker -d --storage-opt dm.loopmetadatasize=4G
```

`dm.fs` 设定文件系统基础设备类型，支持的类型是ext4和xfs，默认是ext4

使用方法：

$ docker -d --storage-opt dm.fs=xfs

`dm.mkfsarg` 设定在创建基础设备时mkfs所用到的参数

使用方法：

```
$ docker -d --storage-opt "dm.mkfsarg=-O ^has_journal"
```

`dm.mountopt` 挂载设备时设置挂载选项。

使用方法：

```
$ docker -d --storage-opt dm.mountopt=nodiscard
```

`dm.blocksize` 为thin pool 设置块大小。默认是64K

使用方法：

$ docker -d --storage-opt dm.blocksize=512K

`dm.blkdiscard` 当删除devicemapper设备时允许或禁止使用blkdiscard 默认是允许（enable）。如果禁止，将会时删除容器更加快速，但是不会返回其中文件的使用空间。

使用说明：

```
$ docker -d --storage-opt dm.blkdiscard=false
```

`dm.override_udev_sync_check` 设置该参数为true，可以协调devicemapper 与 udev的资源利用。当其设置为false时，将会在devicemapper与udev产生竞争，有可能导致错误或者失败。

使用方法：

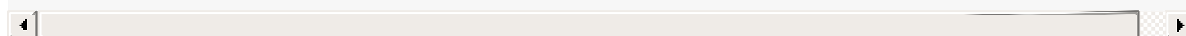
```
$ docker -d --storage-opt dm.override_udev_sync_check=true
```

- Docker execdriver选项

目前zfs支持的选项 `zfs.fsname`

使用方法：

```
$ docker daemon -s zfs --storage-opt zfs.fsname=zroot/docker
```



另外，可以使用 `-e lxc` 来启用 `lxcexecution` 设备

- Daemon DNS选项

设置dns 服务器

```
$  docker -d --dns 8.8.8.8
$  docker -d --dns-search example.com
```

- 不安全仓库登记

一个安全的私有仓库通过使用TLS和CA证书的副本来替换 `/etc/docker/certs.d/myregistry:5000/ca.crt` 文件。不使用TLS，或者使用未知CA证书的TLS都将是不安全的。如果CA证书验证实效或者在 `/etc/docker/certs.d/myregistry:5000/` 找不到证书将会报错。使用 `--insecure-registry` 参数可以标记一个不安全的仓库：

```
--insecure-registry myregistry:5000
```

将告诉 deamon 这个 `myregistry:5000` 仓库应该标记为不安全状态。

```
--insecure-registry 10.1.0.0/16
```

告诉deamon通过CIDR语法解析出来的IP地址是 `10.1.0.0/16` 的仓库标记为不安全。

如果没有使用参数 `--insecure-registry` 标记，那么 `docker pull 、docker push、docker search` 从指定仓库执行时将会报错。

atttach

- 用法

```
Usage: docker attach [OPTIONS] CONTAINER

Attach to a running container

  --help=false        Print usage
  --no-stdin=false    Do not attach STDIN
  --sig-proxy=true    Proxy all received signals to the proces
```

- 例子

在使用-d参数时，容器启动后会进入后台。 某些时候需要进入容器进行操作，有很多种方法，包括使用 docker attach命令或 nsenter 工具等。

```
$  sudo docker run -i -t -d centos
911207826f4da78cb8b8a233dea6120a7d2939eea389a94eef2c0b1320572628
$  sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATEI
911207826f4d        centos:latest       "/bin/bash"         46 seco
$ sudo docker attach 911207826f4d
[root@911207826f4d /]#
[root@911207826f4d /]#
```

此时，我们以及进入一个正在运行的容器中去执行命令。

- 总结

使用 attach 命令有时候并不方便。当多个窗口同时 attach 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时,其他窗口也无法执行操作了。

扩展工具 nsenter

- 说明

nsenter可以访问一个进程大名字空间。 指令时包含在untli-linux（2.23版本之后才会包含）软件包里。这里需要安装一下：

```
$ sudo yum -y install util-linux
```

完成后检验：

```
$ nsenter -V
nsenter from util-linux 2.23.2
```

如果要进入容器内需要知道进程的pid。

```
$ sudo  docker run -idt --name=test1 centos f629fa879a34af902a259e8
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
f629fa879a34        centos              "/bin/bash"         6 secon
$ sudo PID=$(docker inspect --format "{{ .State.Pid }}" f629fa879a3
$ sudo  echo $PID
22109
$ sudo nsenter --target 22109 --mount --uts --ipc --net --pid
[root@f629fa879a34 /]#
[root@f629fa879a34 /]#
[root@f629fa879a34 /]#
```

这样就完成了进入容器内访问的目的。

此外，为了方便进入容器，牛人已经为我们封装好指令，我们只需利用简单的两行代码就可完成操作。 下载这个脚本.bashrc_docker，并将内容放到 .bashrc 中：

```
$ sudo wget -P ~ https://github.com/yeasy/docker_practice/raw/maste
$ sudo echo "[ -f ~/.bashrc_docker ] && . ~/.bashrc_docker" >> ~/.b
```

执行完后我们只需：

```
$ sudo echo $(docker-pid f629fa879a34)
22109
$ sudo docker-enter f629fa879a34
[root@f629fa879a34 ~]#
实际上也是使用nsenter进入容器，只不过更简洁罢了。
```

build

- 用法

```
Usage: docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

  -f, --file=""              Name of the Dockerfile (Default is 'PATH/
  --force-rm=false           Always remove intermediate containers
  --no-cache=false           Do not use cache when building the image
  --pull=false               Always attempt to pull a newer version of
  -q, --quiet=false          Suppress the verbose output generated by
  --rm=true                  Remove intermediate containers after a su
  -t, --tag=""               Repository name (and optionally a tag) fo
  -m, --memory=""            Memory limit for all build containers
  --memory-swap=""           Total memory (memory + swap), `-1` to dis
  -c, --cpu-shares           CPU Shares (relative weight)
  --cpuset-mems=""           MEMs in which to allow execution, e.g. `0
  --cpuset-cpus=""           CPUs in which to allow execution, e.g. `0
  --cgroup-parent=""         Optional parent cgroup for the container
  --ulimit=[]                Ulimit options
```

- 例子

使用该命令，将会从参数指定的路径中的 Dockerfile的文件执行构建镜像，文件的指向可以是一个本地文件PATH或者是一个URL。

例如：

```
$ sudo docker build https://github.com/docker/rootfs.git#container:
```

或者用标准输入：

```
$ sudo docker build - < Dockerfile
```

如果你采用以上两种方式构建镜像，-f 或者－file参数将失效。

默认情况下，docker build 指令将会在指定根目录下查找Dockerfile文件，如果指定-f/-file参数，将指定该构建目录文件，这样的好处是可以多次构建。需要注意的是，路径必须包含构建信息的文件。

在多数情况下，最好保证构建目录为空。然后添加所需要的软件包到该文件夹。为了提高构建效率，可以加入 .dockerignore 文件排除一些不需要的文件。

返回值

如果构建成功，将会返回0，当失败时，将会返回相应错误返回值：

```
$ docker build -t fail .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM busybox
 ---> 4986bf8c1536
Step 1 : RUN exit 13
 ---> Running in e26670ec7a0a
INFO[0000] The command [/bin/sh -c exit 13] returned a non-zero coc
$ echo $?
1
```

一般例子：

```
$ docker build .
Uploading context 10240 bytes
Step 1 : FROM busybox
Pulling repository busybox
 ---> e9aa60c60128MB/2.284 MB (100%) endpoint: https://cdn-registry
Step 2 : RUN ls -lh /
 ---> Running in 9c9e81692ae9
total 24
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 bin
drwxr-xr-x    5 root     root        4.0K Oct 19 00:19 dev
drwxr-xr-x    2 root     root        4.0K Oct 19 00:19 etc
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 lib
lrwxrwxrwx    1 root     root           3 Mar 12  2013 lib64 -> lib
dr-xr-xr-x  116 root     root           0 Nov 15 23:34 proc
lrwxrwxrwx    1 root     root           3 Mar 12  2013 sbin -> bin
dr-xr-xr-x   13 root     root           0 Nov 15 23:34 sys
drwxr-xr-x    2 root     root        4.0K Mar 12  2013 tmp
drwxr-xr-x    2 root     root        4.0K Nov 15 23:34 usr
 ---> b35f4035db3f
Step 3 : CMD echo Hello world
 ---> Running in 02071fceb21b
 ---> f52f38b7823e
Successfully built f52f38b7823e
Removing intermediate container 9c9e81692ae9
Removing intermediate container 02071fceb21b
```

上面例子中，指定路径是 .,这个路径告诉docker构建的目录为当前目录，里面包含构建文件的信息，以及所要添加的文件。如果想保留构建过程中的容器，可以使用--rm=false， 这样操作不会影响构建缓存。

下面这个例子使用了.dockerignore文件来排除.git文件的使用方法，将会影响上下文文件大小。

```
$ docker build .
Uploading context 18.829 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
    $ echo ".git" > .dockerignore
$ docker build .
Uploading context  6.76 MB
Uploading context
Step 0 : FROM busybox
 ---> 769b9341d937
Step 1 : CMD echo Hello world
 ---> Using cache
 ---> 99cc1ad10469
Successfully built 99cc1ad10469
```

使用-t参数指定name以及tag：

```
$ docker build -t vieux/apache:2.0 .
```

从标准输入读取Dockerfile：

```
$ docker build - < Dockerfile
```

使用压缩文件，目前支持的格式是bzip2, gzip and xz

```
$ docker build - < context.tar.gz
```

从克隆的GitHub仓库作为上下文构建镜像，在仓库根目录下的Dockerfile文件将作为构建文件。

```
$ docker build github.com/creack/docker-firefox
```

注意，若要加前缀必须是 git:// 或者 git@ 。

使用-f参数指定文件构建

```
$ docker build -f Dockerfile.debug .
```

在.目录下从不同文件构建镜像：

```
$ docker build -f dockerfiles/Dockerfile.debug -t myapp_debug .
$ docker build -f dockerfiles/Dockerfile.prod  -t myapp_prod .
```

我们在观察下面例子：

```
$ cd /home/me/myapp/some/dir/really/deep
$ docker build -f /home/me/myapp/dockerfiles/debug /home/me/myapp
$ docker build -f ../../../../dockerfiles/debug /home/me/myapp
```

这个例子执行的两次构建操作所做事情是一模一样的，都会寻找debug文件作为Dockerfile来构建镜像。

commit

- 用法

```
Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

-a, --author=       Author (e.g., "John Hannibal Smith <hannibal@a
-c, --change=[]     Apply Dockerfile instruction to the created ima
--help=false        Print usage
-m, --message=      Commit message
-p, --pause=true    Pause container during commit
```

- 例子

```
$ sudo docker ps
ID                  IMAGE               COMMAND             CREATED
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days
197387f1b436        ubuntu:12.04        /bin/bash           7 days
$ sudo docker commit c3f279d17e0a  SvenDowideit/testimage:version3
f5283438590d
$ sudo docker images | head
REPOSITORY                        TAG                 ID
SvenDowideit/testimage            version3            f5283438590d
```

提交一个重新配置过的容器到镜像

```
$ sudo docker ps
ID                  IMAGE               COMMAND             CREATEI
c3f279d17e0a        ubuntu:12.04        /bin/bash           7 days
197387f1b436        ubuntu:12.04        /bin/bash           7 days
$ sudo docker inspect -f "{{ .Config.Env }}" c3f279d17e0a
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
$ sudo docker commit --change "ENV DEBUG true" c3f279d17e0a SvenDov
f5283438590d
$ sudo docker inspect -f "{{ .Config.Env }}" f5283438590d
[HOME=/ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
```

- 总结

这个命令的用处在于把有修改的container提交成新的Image，然后导出此Imange分发给其他场景中调试使用。Docker官方的建议是，当你在调试完Image的问题后，应该写一个新的Dockerfile文件来维护此Image。commit命令仅是一个临时创建Imange的辅助命令。

cp

- 用法

```
Usage: docker cp [OPTIONS] CONTAINER:PATH HOSTDIR|-

Copy files/folders from a PATH on the container to a HOSTDIR on the

  --help=false       Print usage
```

- 例子

```
$ sudo docker cp hopeful_feynman:/etc /home
```

这将会在主机的/home目录下多一个etc文件夹，该文件夹就是从容器中复制出来的。

- 总结

使用cp可以把容器內的文件复制到Host主机上。这个命令在开发者开发应用的场景下，会需要把运行程序产生的结果复制出来的需求，在这个情况下就可以使用这个cp命令。

diff

- 用法

```
Usage: docker diff [OPTIONS] CONTAINER

Inspect changes on a container's filesystem

  --help=false       Print usage
```

- 例子

```
$ sudo docker diff b448f729a0b0
C /run
A /run/secrets
```

- 总结

diff会列出3种容器内文件状态变化（A - Add, D - Delete, C - Change ）的列表清单。构建Image的过程中需要的调试指令。

events

- 用法

```
Usage: docker events [OPTIONS]

Get real time events from the server

-f, --filter=[]    Filter output based on conditions provided
--help=false       Print usage
--since=           Show all events created since timestamp
--until=           Stream events until this timestamp
```

- 例子

第一个窗口用来监听事件

```
$ docker events
```

第二个窗口 起停容器

```
$ docker start 4386fb97867d
$ docker stop 4386fb97867d
$ docker stop 7805c1d35632
```

执行完后，shell窗口会同步打印如下信息：

```
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
```

使用since参数按时间筛选

```
$ sudo docker events --since 1378216169
2014-03-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-03-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ sudo docker events --since '2013-09-03'
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-09-03T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ sudo docker events --since '2013-09-03T15:49:29'
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8)
```

只保留三分钟内的事件

```
$ sudo docker events --since '3m'
2015-05-12T11:51:30.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2015-05-12T15:52:12.999999999Z07:00 4 4386fb97867d: (from ubuntu-1:
2015-05-12T15:53:45.999999999Z07:00  7805c1d35632: (from redis:2.8)
2015-05-12T15:54:03.999999999Z07:00  7805c1d35632: (from redis:2.8)
```

也可以使用过滤器筛选

```
$ docker events --filter 'event=stop'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-09-03T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ docker events --filter 'image=ubuntu-1:14.04'
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
$ docker events --filter 'container=7805c1d35632'
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ docker events --filter 'container=7805c1d35632' --filter 'contair
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ docker events --filter 'container=7805c1d35632' --filter 'event=s
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8)
$ docker events --filter 'container=container_1' --filter 'containe
2014-09-03T15:49:29.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 4386fb97867d: (from ubuntu-1:14
2014-05-10T17:42:14.999999999Z07:00 7805c1d35632: (from redis:2.8)
2014-09-03T15:49:29.999999999Z07:00 7805c1d35632: (from redis:2.8)
```

- 总结

  打印容器实时的系统事件。

export

- 用法

```
Usage: docker export [OPTIONS] CONTAINER

Export a filesystem as a tar archive (streamed to STDOUT by default

  --help=false       Print usage
  -o, --output=      Write to a file, instead of STDOUT
```

- 例子

```
$ sudo docker export b448f729a0b0 > centos.tar
```

- 总结

把容器系统文件打包并导出来，方便分发给其他场景使用。

import

- 用法

```
Usage: docker import [OPTIONS] URL|- [REPOSITORY[:TAG]]

Create an empty filesystem image and import the contents of the
tarball (.tar, .tar.gz, .tgz, .bzip, .tar.xz, .txz) into it, then
optionally tag it.

 -c, --change=[]    Apply Dockerfile instruction to the created ima
  --help=false       Print usage
```

- 例子

从网络上导入：

```
$ sudo docker import http://example.com/exampleimage.tgz
```

从本地文件导入:

通过标准输入和pipe导入到docker.

```
$ cat exampleimage.tgz | sudo docker import - exampleimagelocal:new
```

从本地目录导入：

```
$ sudo tar -c . | docker import - exampleimagedir
```

带配置信息从本地目录导入：

```
$ sudo tar -c . | docker import --change "ENV DEBUG true" - example
```

- 总结

暂无

history

- 用法

```
Usage: docker history [OPTIONS] IMAGE

Show the history of an image

-H, --human=true      Print sizes and dates in human readable format
--help=false          Print usage
--no-trunc=false      Don't truncate output
-q, --quiet=false     Only show numeric IDs
```

- 例子

```
$ sudo docker history postgres
IMAGE               CREATED             CREATED BY
730d1d72bda2        4 weeks ago         /bin/sh -c #(nop) CMD ["pos
3e840dbb5474        4 weeks ago         /bin/sh -c #(nop) EXPOSE 54
4df8a54cf33a        4 weeks ago         /bin/sh -c #(nop) ENTRYPOIN
09e02a9f8afe        4 weeks ago         /bin/sh -c #(nop) COPY file
39172f8b90f2        4 weeks ago         /bin/sh -c #(nop) VOLUME [/
3fa84fbfdec9        4 weeks ago         /bin/sh -c #(nop) ENV PGDAT
c5d75e7f9094        4 weeks ago         /bin/sh -c #(nop) ENV PATH=
a95070c23e86        4 weeks ago         /bin/sh -c mkdir -p /var/ru
64957633c267        4 weeks ago         /bin/sh -c apt-get update &
a814508841fa        4 weeks ago         /bin/sh -c echo 'deb http:/
49915906faae        4 weeks ago         /bin/sh -c #(nop) ENV PG_VE
b41b53da5fba        4 weeks ago         /bin/sh -c #(nop) ENV PG_MA
02fa71f1fa38        4 weeks ago         /bin/sh -c apt-key adv --ke
0b82f508e063        4 weeks ago         /bin/sh -c mkdir /docker-en
e07b5a739ed9        4 weeks ago         /bin/sh -c #(nop) ENV LANG=
c783ebe7a1d4        4 weeks ago         /bin/sh -c apt-get update &
8b6b2a3b7f9c        4 weeks ago         /bin/sh -c apt-get update &
22ed955cce18        5 weeks ago         /bin/sh -c gpg --keyserver
26a84c436db4        5 weeks ago         /bin/sh -c groupadd -r post
9a61b6b1315e        5 weeks ago         /bin/sh -c #(nop) CMD ["/bi
902b87aaaec9        5 weeks ago         /bin/sh -c #(nop) ADD file:
```

- 总结

打印指定Image中每一层Image命令行的历史记录。

images

- 使用方法

```
docker images [OPTIONS] [REPOSITORY]

List images

 -a, --all=false      Show all images (default hides intermediate i
  --digests=false      Show digests
  -f, --filter=[]      Filter output based on conditions provided
  --help=false         Print usage
  --no-trunc=false     Don't truncate output
  -q, --quiet=false    Only show numeric IDs
```

- 例子：

查询本里存储的镜像

```
$ sudo docker imgaes
REPOSITORY           TAG       IMAGE ID       CREATED       VIRTUAL
docker.io/ubuntu     latest    63e3c10217b8   7 days ago     188.3
docker.google/etcd   2.1.1     2c319269dd15   8 days ago     23.32
docker.io/postgres   latest    730d1d72bda2   2 weeks ago    265.3
centos               latest    770327a1e9e7   2 weeks ago    418.9
…
```

将ID完整展现

```
$ sudo docker images --no-trunc
REPOSITORY                  TAG                   IMAGE ID
scratch1                    latest                dc869bfd3085af05a1a
registry.liugang/centos     latest                770327a1e9e746cf8d4
registry.liugang/busybox    latest                8c2e06607696bd4afb3
docker.io/scratch           latest                511136ea3c5a64f264b
```

使用该命令将展现没有tag的镜像

```
$ sudo docker images --filter "dangling=true"
REPOSITORY          TAG                 IMAGE ID            CREATED
<none>              <none>              b133995b6291        About a
<none>              <none>              6fae83243a01        About a
<none>              <none>              4c6412305cfa        About a
```

- 总结

其中第一字段是image镜像的名称；TAG一般表示为版本号，也可以自己定义 ；IMAGE ID 表示镜像的唯一ID， 这也是判断两个镜像文件是否为同一个的判断标准。

info

- 用法

```
Usage: docker info [OPTIONS]

Display system-wide information

  --help=false       Print usage
```

- 例子

```
$ sudo docker -D info
Containers: 6
Images: 30
Storage Driver: devicemapper
 Pool Name: docker-8:3-28326-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: xfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 1.37 GB
 Data Space Total: 107.4 GB
 Data Space Available: 44.49 GB
 Metadata Space Used: 2.245 MB
 Metadata Space Total: 2.147 GB
 Metadata Space Available: 2.145 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: false
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/meta
 Library Version: 1.02.93-RHEL7 (2015-01-28)
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.10.0-229.el7.x86_64
Operating System: CentOS Linux 7 (Core)
CPUs: 1
Total Memory: 979.7 MiB
Name: localhost.localdomain
ID: PRVB:3SDE:YL4E:JT5P:5BIR:BUC5:PHXI:HG4B:P753:Y2BI:U7OU:YPGC
```

- 总结

这个命令在开发者报告Bug时会非常有用，结合docker vesion一起，可以随时使用这个命令把本地的配置信息提供出来，方便Docker的开发者快速定位问题。

inspect

- 用法

```
Usage: docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...

Return low-level information on a container or image

-f, --format=          Format the output using the given go template
--help=false           Print usage
-r, --remote=false     Inspect remote images
```

- 例子

```
$ sudo docker inspect centos
[
{
    "Id": "770327a1e9e746cf8d4449a7134e87917982b33c7f5cea584d941350
    "Parent": "67c02c69a0fc420e781b9a1c676f19306e999aac2cf3ba24dfa4
    "Comment": "",
    "Created": "2015-07-24T01:06:38.020790544Z",
    "Container": "9163378b9f7fe2887bce56cb726c3845ce9af8ebc9cbef30c
"ContainerConfig": {
    "Hostname": "9163378b9f7f",
    "Domainname": "",
    "User": "",
    "AttachStdin": true,
     ...
     ...
     ...
    }
```

取出某一个值：

```
$ sudo docker inspect --format="{{.Id}}" centos
770327a1e9e746cf8d4449a7134e87917982b33c7f5cea584d941350f5ead7ac
```

- 总结

查看容器运行时详细信息的命令。了解一个Image或者Container的完整构建信息就可以通过这个命令实现。

login

- 用法

```
Usage: docker login [OPTIONS] [SERVER]

Register or log in to a Docker registry server, if no server is
specified "https://index.docker.io/v1/" is the default.

-e, --email=       Email
--help=false       Print usage
-p, --password=    Password
-u, --username=    Username
```

- 例子

```
root@liugang:~# docker login
Username: username
Password: ****
Email: user@domain.com
Login Succeeded
```

如果你有一个自己的仓库，你也可以连接到指定主机：

```
$ sudo docker login localhost:8080
```

logout

- 用法

```
Usage: docker logout [OPTIONS] [SERVER]

Log out from a Docker registry, if no server is
specified "https://index.docker.io/v1/" is the default.

--help=false       Print usage
```

- 例子

```
$ sudo docker logout localhost:8080
```

logs

- 用法

```
Usage: docker logs [OPTIONS] CONTAINER

Fetch the logs of a container

-f, --follow=false        Follow log output
--help=false              Print usage
--since=                  Show logs since timestamp
-t, --timestamps=false    Show timestamps
--tail=all                Number of lines to show from the end of t
```

- 例子

```
$ sudo docker logs 60095325e584
```

- 总结

批量打印出容器中进程的运行日志。

network(1.9.0版本)

network connect

你可以使用容器名称或者ID，将一个正在运行的容器介入网络。连接成功后容器酒可以与处于同一个网络中的容器通信。

```
docker network connect multi-host-network coantainer1
```

你也可以使用 docker run --net= 选项来启动一个容器直接连接到一个已知网络。

```
docker run -idt --net=multi-host-network busybox
```

你可以暂停，重启甚至终止已连接网络的容器。暂停容器将保持网络连接以及网络发现（ by a network inspect）。终止容器，将使容器在该网络上消失，直到重启才可以被发现。当容器重启以后，容器重新加入该网络将不保证IP地址保持与原来一致。

使用 docker network inspect 命令来验证容器的网络是否已连接，而使用 docker network disconnect 来从网络上断开与容器的连接、

当容器连接到网络时，容器职能使用容器IP地址或者容器name来通信。对于overlay网络或者其它通过插件配置的跨主机环境的网络，然可以使用这种方式运行。

你可以使容器连接一个活多个网络，这些网络不要是相同类型的。例如：你可以连接一个容器网桥和overlay网络。

network create

该命令用于创建一个网络。使用 -d参数允许使用bridge或者overlay类型的网络构建与网络驱动中。如果你有第三种网络结构或者其它通用网络驱动，你也可以使用该参数特别说明。

如果不指定 --driver 参数，该命令将自动为你创建一个bridge类型的网络。该网络对应与传统的docker0网桥。当使用 docker run 启动一个容器时，它将自动连接到这个bridge网络。你不能删除这个默认的网络但是可以使用docker network create 命令创建一个新的：

```
docker network create -d bridge my-bridge-network
```

birdge网络是单docker引擎的隔离网络（Bridge networks are isolated networks on a single Engine installation）。如果你想创建一个跨越多个docker主机引擎的网络，你必须创建一个overlay类型的网络。与birdge网络不同，overlay网络创建需要提前准备一些配置：

```
1、需要连接一个 key－value 存储，目前支持Consul，Etcd以及Zookeeper（分布式

2、一个连接到 key－value 存储的云主机

3、每一台机器上的 docker deamon 都要配置相同参数
```

docker deamon 支持overlay选项的参数有：

```
--cluster-store
--cluster-store-opt
--cluster-advertise
```

想要了解更多有关配置以上参数的信息，请阅读 “Get started with multi-host network“

你可以安装 docker swarm 来管理集群建立自己的网络，这是一个不错的想法，但不是必须的。 Swarm提供先进的服务发现以及节点管理来帮助你实现。

当你准备好你的overlay网络时，你只需选择集群中的一个docker主机并执行以下命令：

```
docker network create -d overlay my-multihost-network
```

网络名称必须是唯一的，docker deamon 会尝试验证命名冲突，但并不保证（我就呵呵了）。用户有责任去避免命名冲突（呵呵呵。。。）。

connect containers

当你使用 --net 参数连接到一个网络时，这将会使目标容器连接到自定义网络中去：

```
docker run -idt --net=mynet busybox
```

如果你想在一个容器运行之后添加到一个网络中去，可以使用 docker network connect 命令。

你可以将多个容器连接到相同的网络中去。一旦连接，容器将只能通过IP或者容器名称进行通信。对于overlay网络或者其它通过插件配置的跨主机环境的网络，然可以使用这种方式运行。

使用 docker network disconnect 可以断开容器与网络的连接

Specifying advanced options

当创建一个网络时，docker Engine 会默认为该网络创建一个非重叠子网。这个子网并不是已存在子网的划分，它纯粹为了IP寻址（It is purely for ip-addressing purposes）。你可以覆盖这个默认的，然后使用 --subnet 选项来特别定义。在bridge网络上你可以这样定义：

```
docker network create -d --subnet=192.168.0.0/16
```

此外，你还可以指定 --gateway --ip-range 以及 --aux-addressoptions。

```
docker network create --driver=bridge  --subnet=172.28.0.0/16  --ip
```

如果你省略了 --gateway 选项，docker Engine 将会从内置的 preferred pool 为你选择一个。

对于overlay网络，你可以创建多个子网：

```
docker network create -d overlay
--subnet=192.168.0.0/16 --subnet=192.170.0.0/16
--gateway=192.168.0.100 --gateway=192.170.0.100
--ip-range=192.168.1.0/24
--aux-address a=192.168.1.5 --aux-address b=192.168.1.6
--aux-address a=192.170.1.5 --aux-address b=192.170.1.6
my-multihost-newtork
```

但是确保你的子网不要重叠，否则，创建网络就会失败，Engine 将会反悔错误。

network disconnect

断开容器与网络的连接。

```
 docker network disconnect multi-host-network container1
```

network ls

列出deamon知道的所有的网络，包括跨多主机的集群网络。

```
 sudo docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge                 bridge
9f904ee27bf5        none                    null
cf03ee007fb4        host                      host
78b03ee04fc4        multi-host          overlay
```

使用 --no-trunc 选项来显示整个网络的ID

```
docker network ls --no-trunc
NETWORK ID
18a2866682b85619a026c81b98a5e375bd33e1b0936a26cc497c283d27bae9b3
c288470c46f6c8949c5f7e5099b5b7947b07eabe8d9a27d79a9cbf111adcbf47
7b369448dccbf865d397c8d2be0cda7cf7edc6b0945f77d2529912ae917a0185
95e74588f40db048e86320c6526440c504650a1ff3e9f7d60a497c4d2163e5bd
```

network rm

删除一个网络，在删除该网络之前，必须断开与该网络连接的任何容器。

```
docker network rm my-network
```

search

- 用法

```
Usage: docker search [OPTIONS] TERM

Search the Docker Hub for images

  --automated=false    Only show automated builds
  --help=false         Print usage
  --no-index=false     Don't prepend index to output
  --no-trunc=false     Don't truncate output
  -s, --stars=0        Only displays with at least x stars
```

- 例子

```
$ sudo docker search ubuntu
```

从官方仓库中搜索出含有关键字ubuntu的镜像：

```
INDEX        NAM                                       DESCRIPTION
docker.io    docker.io/ubuntu                          Ubuntu is a D
docker.io    docker.io/ubuntu-upstart                  Upstart is an
docker.io    docker.io/torusware/speedus-ubuntu        Always update
docker.io    docker.io/dorowu/ubuntu-desktop-lxde-vnc  Ubuntu with o
docker.io    docker.io/sequenceiq/hadoop-ubuntu        An easy way t
docker.io    docker.io/tleyden5iwx/ubuntu-cuda         Ubuntu 14.04
docker.io    docker.io/ubuntu-debootstrap              debootstrap -
…
```

- 总结

在使用docker创建容器时，必然要用到镜像文件。这时我们就得从仓库中拉取我们所需要的image文件。

pull

- 用法

```
Usage: docker pull [OPTIONS] NAME[:TAG|@DIGEST]

Pull an image or a repository from the registry

-a, --all-tags=false    Download all tagged images in the repositor
--help=false            Print usage
```

- 例子

找到所需要的镜像：

```
$ sudo docker pull docker.io/ubuntu:12.04
```

这里是从官方仓库中拉取下来版本号(TAG)为12.04的镜像，其中“docker.io” 可以不写，默认是从官方仓库下载。版本号(TAG)不写的话默认会拉取一个版本号为latest的镜像文件：

```
Trying to pull repository docker.io/ubuntu ...
d0e008c6cf02: Download complete
a69483e55b68: Download complete
bc99d1f906ec: Download complete
3c8e79a3b1eb: Download complete
Status: Downloaded newer image for docker.io/ubuntu:12.04
```

见到如上类似结果说明镜像拉取成功。现在看一下自己的仓库，多了一个12.04的镜像。

```
$ sudo docker images
REPOSITORY          TAG       IMAGE ID       CREATED      VIRTUAL SIZ
docker.io/ubuntu    latest    63e3c10217b8   7 days ago   188.3 MB
docker.io/ubuntu    12.04     d0e008c6cf02   7 days ago   134.7 MB
docker.google/etcd 2.1.1     2c319269dd15   8 days ago   23.32 MB
docker.io/postgres latest    730d1d72bda2   2 weeks ago  265.3 MB
…
```

- 总结

push

- 用法

```
Usage: docker push [OPTIONS] NAME[:TAG]

Push an image or a repository to the registry

  -f, --force=false    Push to public registry without confirmation
  --help=false         Print usage
```

- 例子

```
$ sudo docker push docker.io/ubuntu:latest
```

- 总结

镜像的上传，push 默认是向官方仓库上传，由于服务器在国外，传输速度非常慢，就没试验成功过。注意的是，需要在docker hub上注册过后才可以上传镜像哦。关于私有仓库的上传将在后面章节详细讲解。

ps

- 用法

  Usage: docker ps [OPTIONS]

  List containers

  ```
  -a, --all=false       Show all containers (default shows just r
  --before=             Show only container created before Id or
  -f, --filter=[]       Filter output based on conditions provide
  --help=false          Print usage
  -l, --latest=false    Show the latest created container, incluc
  -n=-1                 Show n last created containers, include r
  --no-trunc=false      Don't truncate output
  -q, --quiet=false     Only display numeric IDs
  -s, --size=false      Display total file sizes
  --since=              Show created since Id or Name, include nc
  ```

- 例子

```
$ sudo docker ps -a
CONTAINER ID  IMAGE    COMMAND       CREATED       STATUS
b448f729a0b0  centos   "/bin/bash"   4 days ago    Exited (137) 4 c
54c7b6d6632e  centos   "/bin/bash"   4 days ago    Exited (0) 3 day
```

利用筛选器筛选出exied状态时0的容器：

```
$ sudo docker ps -a --filter 'exited=0'
CONTAINER ID        IMAGE                      COMMAND             C
8d92293a65e9        registry.liugang/centos    "/bin/bash"         7
8410f389ea65        registry.liugang/centos    "/bin/bash"         7
```

- 总结

-a参数列出所有状态的容器， -l列出最新创建的容器，包括停止运行状态的容器。

kill

- 用法

```
Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Kill a running container using SIGKILL or a specified signal

  --help=false          Print usage
  -s, --signal=KILL     Signal to send to the container
```

- 例子

```
$ sudo docker kill pensive_wilson
pensive_wilson
```

这将停止该容器

- 总结

结合ps命令，可以做到kill所有正在运行的容器：

```
$ sudo docker kill $(sudo docker ps -a -q)
```

rm

- 用法

```
Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

  -f, --force=false      Force the removal of a running container (
  --help=false           Print usage
  -l, --link=false       Remove the specified link
  -v, --volumes=false    Remove the volumes associated with the con
```

- 例子

```
$ sudo docker rm pensive_wilson
pensive_wilson
```

这将删除一个已经停止运行的容器，若容器正在运行，则将会使docker报错，停止容器再删除，或者加上-f参数强制删除（不建议）。

- 总结

类似的我们也结合ps删除所有容器：

```
$ sudo docker kill $(sudo docker ps -a -q)
...
...
...
$ sudo docker rm $(sudo docker ps -a -q)
...
...
...
```

要清空容器，首先要保证没有容器在运行。

rmi

- 用法

```
Usage: docker rmi [OPTIONS] IMAGE [IMAGE...]

Remove one or more images

  -f, --force=false     Force removal of the image
  --help=false          Print usage
  --no-prune=false      Do not delete untagged parents
```

- 例子

```
$ sudo docker rmi centos:6.5
...
...
...
```

- 总结

要区分rm于rmi多用法。 与docker images命令配合来清空镜像：

```
$ sudo docker rmi $(sudo docker images -a -q)
```

port

- 用法

```
Usage: docker port [OPTIONS] CONTAINER [PRIVATE_PORT[/PROTO]]

List port mappings for the CONTAINER, or lookup the public-facing p
is NAT-ed to the PRIVATE_PORT

  --help=false       Print usage
```

- 例子

```
$ sudo docker port 60095325e584
```

- 总结

打印出Host主机端口与容器暴露出的端口的NAT映射关系

pause

- 用法

```
Usage: docker pause [OPTIONS] CONTAINER [CONTAINER...]

Pause all processes within a container

  --help=false       Print usage
```

- 例子

```
$ sudo docker pauese hopeful_feynman
hopeful_feynman
CONTAINER ID    IMAGE     COMMAND      CREATED         STATUS
c9a12157fed7    centos    "/bin/bash"  9 minutes ago   Up 9 minutes
```

使容器内进程暂停

unpause

- 用法

```
Usage: docker unpause [OPTIONS] CONTAINER [CONTAINER...]

Unpause all processes within a container

  --help=false       Print usage
```

- 例子

```
$ sudo docker pauese hopeful_feynman
hopeful_feynman
```

恢复暂停

create

- 用法

```
Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]

Create a new container

      -a, --attach=[]               Attach to STDIN, STDOUT or STDERR
      --add-host=[]                 Add a custom host-to-IP mapping (
      --blkio-weight=0              Block IO (relative weight), betwe
      -c, --cpu-shares=0            CPU shares (relative weight)
      --cap-add=[]                  Add Linux capabilities
      --cap-drop=[]                 Drop Linux capabilities
      --cgroup-parent=              Optional parent cgroup for the co
      --cidfile=                    Write the container ID to the fil
      --cpu-period=0                Limit CPU CFS (Completely Fair Sc
      --cpu-quota=0                 Limit the CPU CFS quota
      --cpuset-cpus=                CPUs in which to allow execution
      --cpuset-mems=                MEMs in which to allow execution
      --device=[]                   Add a host device to the containe
      --dns=[]                      Set custom DNS servers
      --dns-search=[]               Set custom DNS search domains
      -e, --env=[]                  Set environment variables
      --entrypoint=                 Overwrite the default ENTRYPOINT
      --env-file=[]                 Read in a file of environment var
      --expose=[]                   Expose a port or a range of ports
      -h, --hostname=               Container host name
      --help=false                  Print usage
      -i, --interactive=false       Keep STDIN open even if not attac
      --init=                       Run container following specified
      --ipc=                        IPC namespace to use
      -l, --label=[]                Set meta data on a container
      --label-file=[]               Read in a line delimited file of
      --link=[]                     Add link to another container
      --log-driver=                 Logging driver for container
      --log-opt=[]                  Log driver options
      --lxc-conf=[]                 Add custom lxc options
      -m, --memory=                 Memory limit
```

```
      --mac-address=              Container MAC address (e.g. 92:d(
      --memory-swap=              Total memory (memory + swap), '-:
      --name=                     Assign a name to the container
      --net=bridge                Set the Network mode for the cont
      --oom-kill-disable=false    Disable OOM Killer
      -P, --publish-all=false     Publish all exposed ports to ranc
      -p, --publish=[]            Publish a container's port(s) to
      --pid=                      PID namespace to use
      --privileged=false          Give extended privileges to this
      --read-only=false           Mount the container's root filesy
      --restart=no                Restart policy to apply when a cc
      --security-opt=[]           Security Options
      -t, --tty=false             Allocate a pseudo-TTY
     -u, --user=                 Username or UID (format: <name|uic
      --ulimit=[]                 Ulimit options
      --uts=                      UTS namespace to use
      -v, --volume=[]             Bind mount a volume
      --volumes-from=[]           Mount volumes from the specified
      -w, --workdir=              Working directory inside the cont
```

- 例子

```
$ sudo docker create ubuntu /bin/echo 'Hello world'
a637c1d67506951928be296f2db02fa3e2b6e974ef371b181d9c26d1c8995963
$...
```

若有如上输出则代表容器创建成功

```
$ sudo docker ps -a
CONTAINER ID        IMAGE                       COMMAND
a637c1d67506        ubuntu:latest               "/bin/echo 'Hello wo
...
```

然后在启动它

```
$ sudo docker start a637c1d67506
a637c1d67506
$...
```

启动成功后会反回容器ID。

```
$ docker ps -a
CONTAINER ID        IMAGE                        COMMAND
a637c1d67506        ubuntu:latest               "/bin/echo 'Hello wo
```

- 总结

当我们去查看容器状态时，容器没有在运行，这时因为我们在创建容器的时候，让容器执行的命令是/bin/echo 'Hello world'，当容器执行完命令的时候就终止结束了。

run

- 用法

```
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

     -a, --attach=[]            Attach to STDIN, STDOUT or STDERR
      --add-host=[]              Add a custom host-to-IP mapping (
      --blkio-weight=0           Block IO (relative weight), betwe
      -c, --cpu-shares=0         CPU shares (relative weight)
      --cap-add=[]               Add Linux capabilities
      --cap-drop=[]              Drop Linux capabilities
      --cgroup-parent=           Optional parent cgroup for the co
      --cidfile=                 Write the container ID to the fil
      --cpu-period=0             Limit CPU CFS (Completely Fair Sc
      --cpu-quota=0              Limit the CPU CFS quota
      --cpuset-cpus=             CPUs in which to allow execution
      --cpuset-mems=             MEMs in which to allow execution
      -d, --detach=false         Run container in background and p
      --device=[]                Add a host device to the containe
```

```
      --dns=[]                        Set custom DNS servers
      --dns-search=[]                 Set custom DNS search domains
      -e, --env=[]                    Set environment variables
      --entrypoint=                   Overwrite the default ENTRYPOINT
      --env-file=[]                   Read in a file of environment var
      --expose=[]                     Expose a port or a range of ports
      -h, --hostname=                 Container host name
      --help=false                    Print usage
      -i, --interactive=false         Keep STDIN open even if not attac
      --init=                         Run container following specified
      --ipc=                          IPC namespace to use
      -l, --label=[]                  Set meta data on a container
      --label-file=[]                 Read in a line delimited file of
      --link=[]                       Add link to another container
      --log-driver=                   Logging driver for container
      --log-opt=[]                    Log driver options
      --lxc-conf=[]                   Add custom lxc options
      -m, --memory=                   Memory limit
      --mac-address=                  Container MAC address (e.g. 92:d0
      --memory-swap=                  Total memory (memory + swap), '-1
      --name=                         Assign a name to the container
      --net=bridge                    Set the Network mode for the cont
      --oom-kill-disable=false        Disable OOM Killer
      -P, --publish-all=false         Publish all exposed ports to rand
      -p, --publish=[]                Publish a container's port(s) to
      --pid=                          PID namespace to use
      --privileged=false              Give extended privileges to this
      --read-only=false               Mount the container's root filesy
      --restart=no                    Restart policy to apply when a co
      --rm=false                      Automatically remove the containe
      --security-opt=[]               Security Options
      --sig-proxy=true                Proxy received signals to the pro
      -t, --tty=false                 Allocate a pseudo-TTY
      -u, --user=                     Username or UID (format: <name|ui
      --ulimit=[]                     Ulimit options
      --uts=                          UTS namespace to use
      -v, --volume=[]                 Bind mount a volume
      --volumes-from=[]               Mount volumes from the specified
      -w, --workdir=                  Working directory inside the cont
```

- 例子

用法与create类似，只是在创建容器后不需要进行start操作就可以运行。

```
$  sudo docker run ubuntu /bin/echo 'Hello world'
Hello world
$...
```

与上面一样，在运行完Hello world 之后也会退出容器。

Daemonized（守护态）

往往我们需要容器在后台一致执行，这时我们就需要在创建镜像的时候让容器以守护台方式(-d 参数)运行。

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello w
61f37c1940c8ec9f08b107e99655b8a5181ded340415e3c15cf413069d556b73
$...
```

这时，我们查看一下容器状态：

```
$ sudo  docker ps -a
CONTAINER ID        IMAGE                   COMMAND                 C
61f37c1940c8        ubuntu:latest           "/bin/sh -c 'while t    4
...
```

查看容器输出的信息

```
$ sudo docker logs 61f37c1940c8
hello world
hello world
hello world
hello world
...
```

- 总结

让容器以后台方式运行，并不是加一个 -d 参数就可以，命令行COMMAND所执行的动作必须为持续运行的状态。

save

- 用法

```
Usage: docker save [OPTIONS] IMAGE [IMAGE...]

Save an image(s) to a tar archive (streamed to STDOUT by default)

  --help=false       Print usage
  -o, --output=      Write to an file, instead of STDOUT
```

- 例子

载出镜像到文件

```
$ sudo docker save -o /home/ubuntu.tar  docker.io/ubuntu:latest
```

这样我们就在/home目录下找到ubuntu.tar 文件了

- 总结

在天朝，docker.io的下载速度奇慢，基本上下一个500M的image就可以搞你半天时间，这时我们就可以利用载入载出，从好朋友那里获取我们需要的镜像啦！;)

load

- 用法

```
Usage: docker load [OPTIONS]

Load an image from a tar archive on STDIN

  --help=false       Print usage
  -i, --input=       Read from a tar archive file, instead of STDIN
```

- 例子

从文件载入镜像到本地

```
$ sudo docker load --input /home/ubuntu.tar
```

或者

```
$ sudo docker load < /home/ubuntu.tar
```

- 总结

这种方式将导入镜像以及其相关的元数据信息（包括标签等）。

start

- 用法

```
Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]

Start one or more stopped containers

-a, --attach=false         Attach STDOUT/STDERR and forward signals
--help=false               Print usage
-i, --interactive=false    Attach container's STDIN
```

stop

- 用法

```
Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]

Stop a running container by sending SIGTERM and then SIGKILL after
grace period

--help=false       Print usage
-t, --time=10      Seconds to wait for stop before killing it
```

restart

- 用法

```
Usage: docker restart [OPTIONS] CONTAINER [CONTAINER...]

Restart a running container

--help=false       Print usage
-t, --time=10      Seconds to wait for stop before killing the cont
```

stats

- 用法

```
Usage: docker stats [OPTIONS] CONTAINER [CONTAINER...]

Display a live stream of one or more containers' resource usage sta

--help=false          Print usage
--no-stream=false     Disable streaming stats and only pull the firs
```

- 例子

```
$ docker stats redis1 redis2
CONTAINER           CPU %               MEM USAGE/LIMIT     MEM %
redis1              0.07%               796 KB/64 MB        1.21%
redis2              0.07%               2.746 MB/64 MB      4.29%
```

- 总结

该指令将只返回运行状态容器的数据流情况，停止状态的容器将不会返回任何数据。

tag

- 用法

```
Usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]N
           docker tag -l [REGISTRYHOST/][USERNAME/]NAME...

Tag an image or list remote tags

-f, --force=false     Force
--help=false          Print usage
-l, --list=false      List repository tags
-r, --remote=false    Force listing of remote repositories only
```

- 例子

```
$ sudo docker tag docker.io/scratch:latest  local/scratch:my
```

- 总结

组合使用用户名，Image名字，标签名来组织管理Image。

top

- 用法

```
Usage: docker top [OPTIONS] CONTAINER [ps OPTIONS]

Display the running processes of a container

--help=false       Print usage
```

- 例子

运行一个之前的例子：

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello w
```

然后执行docker top 命令，可以查看到容器内进程

```
$ sudo docker top 52c058ff716d
UID                 PID                 PPID                C
root                6326                1489                0
root                6410                6326                0
```

wait

- 用法

```
Usage: docker wait [OPTIONS] CONTAINER [CONTAINER...]

Block until a container stops, then print its exit code.

--help=false       Print usage
```

- 说明

阻塞对指定容器的其他调用方法，直到容器停止后退出阻塞。

docker run 的常用参数用法

Docker run 作为运行容器的直接入口，命令参数相当丰富，使用它可以启动容器，使容器拥有自己的文件系统、网络以及关系进程树。

Docker run 命令基本结构：

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

为了更好理解，我们将参数分为以下几类：

1. 容器管理：
    - 后台程序和前台交互程序
    - 器的定义
2. 网络设置
3. CPU和内存的runtime
4. 权限和LXC配置

```
-a, --attach=[]              Attach to STDIN, STDOUT or STDERR
--add-host=[]                Add a custom host-to-IP mapping (host:i
--blkio-weight=0             Block IO (relative weight), between 10
-c, --cpu-shares=0           CPU shares (relative weight)
--cap-add=[]                 Add Linux capabilities
--cap-drop=[]                Drop Linux capabilities
--cgroup-parent=             Optional parent cgroup for the containe
--cidfile=                   Write the container ID to the file
--cpu-period=0               Limit CPU CFS (Completely Fair Schedule
--cpu-quota=0                Limit the CPU CFS quota
--cpuset-cpus=               CPUs in which to allow execution (0-3,
--cpuset-mems=               MEMs in which to allow execution (0-3,
-d, --detach=false           Run container in background and print c
--device=[]                  Add a host device to the container
--dns=[]                     Set custom DNS servers
--dns-search=[]              Set custom DNS search domains
-e, --env=[]                 Set environment variables
--entrypoint=                Overwrite the default ENTRYPOINT of the
--env-file=[]                Read in a file of environment variables
--expose=[]                  Expose a port or a range of ports
```

```
-h, --hostname=              Container host name
--help=false                 Print usage
-i, --interactive=false      Keep STDIN open even if not attached
--init=                      Run container following specified init
--ipc=                       IPC namespace to use
-l, --label=[]               Set meta data on a container
--label-file=[]              Read in a line delimited file of labels
--link=[]                    Add link to another container
--log-driver=                Logging driver for container
--log-opt=[]                 Log driver options
--lxc-conf=[]                Add custom lxc options
-m, --memory=                Memory limit
--mac-address=               Container MAC address (e.g. 92:d0:c6:0a
--memory-swap=               Total memory (memory + swap), '-1' to d
--name=                      Assign a name to the container
--net=bridge                 Set the Network mode for the container
--oom-kill-disable=false     Disable OOM Killer
-P, --publish-all=false      Publish all exposed ports to random por
-p, --publish=[]             Publish a container's port(s) to the ho
--pid=                       PID namespace to use
--privileged=false           Give extended privileges to this contai
--read-only=false            Mount the container's root filesystem a
--restart=no                 Restart policy to apply when a containe
--rm=false                   Automatically remove the container when
--security-opt=[]            Security Options
--sig-proxy=true             Proxy received signals to the process
-t, --tty=false              Allocate a pseudo-TTY
-u, --user=                  Username or UID (format: <name|uid>[:<g
--ulimit=[]                  Ulimit options
--uts=                       UTS namespace to use
-v, --volume=[]              Bind mount a volume
--volumes-from=[]            Mount volumes from the specified contai
-w, --workdir=               Working directory inside the container
```

守护态运行 Detached

当我们启动一个container时，首先需要确定这个container是运行在前台模式还是运行在后台模式。

如果在docker run 后面追加-d=true或者-d，则containter将会运行在后台模式(Detached mode)。此时所有I/O数据只能通过网络资源或者共享卷组来进行交互。因为container不再监听你执行docker run的这个终端命令行窗口。正如之前的例子：

```
$ sudo docker run -d ubuntu /bin/sh -c "while true; do echo hello w
61f37c1940c8ec9f08b107e99655b8a5181ded340415e3c15cf413069d556b73
$...
```

但你可以通过执行docker attach 来重新挂载这个container里面。

```
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                CREA
0409679f511a        ubuntu              "/bin/sh -c 'while t   5 se
$ sudo docker attach 0409679f511a
hello world
hello world
hello world
…
```

需要注意的是，如果你选择执行*-d*使*container*进入后台模式，那么将无法配合*"--rm"*参数。

带控制窗口运行

与Detached（—d）对应的是Foregroud

如果在docker run后面没有追加-d参数，则container将默认进入前台模式(Foregroud mode)。Docker会启动这个container，同时将当前的命令行窗口挂载到container的标准输入，标准输出和标准错误中。也就是container中所有的输出，你都可以再当前窗口中查看到。甚至docker可以虚拟出一个TTY窗口，来执行信号中断。这一切都是可以配置的：

```
-a=[]                 : Attach to `STDIN`, `STDOUT` and/or `STDERR`
-t=false              : Allocate a pseudo-tty
--sig-proxy=true      : Proxify all received signal to the process (r
-i=false              : Keep STDIN open even if not attached
```

如果在执行run命令时没有指定-a，那么docker默认会挂载所有标准数据流，包括输入输出和错误。你可以特别指定挂载哪个标准流。

```
$ sudo docker run -a stdin -a stdout -i -t ubuntu /bin/bash (只挂载标
```

对于执行容器内的交互式操作，例如shell脚本。我们必须使用 -i -t来申请一个控制台同容器进行数据交互。但是当通过管道与容器进行交互时，就不能使用-t. 例如下面的命令：

```
$ echo test | docker run -i busybox cat
```

若强行加上-t 就会报出cannot enable tty mode on non tty input错误。

给容器命名

给container 命名有三种方式：

1. 使用UUID长命("f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e74778")，在创建容器时返回的id就是这个。
2. 使用UUID短命令("f78375b1c487")，当执行查询时，查到的dockerID就是这个。
3. 使用--name=evil_ptolemy",若不加此指令，docker会自动给新创建出来的容器分配一个唯一的name

```
$ sudo docker run -d --name=test_name registry.liugang/centos:
f78375b1c487e03c9438c729345e54db9d20cfa2ac1fc3494b6eb60872e747
$ sudo docker ps -a
CONTAINER ID        IMAGE                             COMMAND
f78375b1c487        registry.liugang/centos:latest    "/bin/bas
```

清除容器

Clean up (--rm) 指在容器运行完之后自动清除

```
--rm=false: Automatically remove the container when it exits (incor
```

默认情况下，每个container在退出时，它的文件系统也会保存下来。这样一方面调试会方便些，因为你可以通过查看日志等方式来确定最终状态。另外一方面，你也可以保存container所产生的数据。但是当你仅仅需要短期的运行一个前台container，这些数据同时不需要保留时。你可能就希望docker能在container结束时自动清理其所产生的数据。 这个时候你就需要--rm这个参数了。

```
$ sudo docker run --rm centos:latest
$ docker ps -a
CONTAINER ID        IMAGE        COMMAND     CREATED      STATUS
(无)
```

注意：*--rm* 和 *-d*不能共用！

数据管理

通常情况下，我们是不会在容器中存储数据的。我们会挂载一个主机的文件夹作为数据卷到容器中去，该数据卷可以被许多需要访问的容器访问得到。这样做的好处是保证了数据的持久化，也增强来应用的可移植性（不改变容器配置）。

Docker内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

```
-v=[]:                   Create a bind mount with: [host-dir]:[conta
                         If "container-dir" is missing, then docker c
--volumes-from="":       Mount all volumes from the given container(s
```

即：数据卷（Data volumes）数据卷容器（Data volume containers）

数据卷

参数的作用就是挂载一个文件目录到指定容器中去，实现容器中数据持久化。

- 数据卷是一个可以供一个或多个使用的特殊目录，它绕过UFS，可以提供很多有用的特性
    - 数据卷可以在容器之间共享和重用
    - 对数据卷的修改会立马生效
    - 对数据卷的更新，不会影响镜像
    - 卷会一直存在，直到没有容器使用

- # 挂载目录

在使用docker run时，加上-v参数可以创建一个数据卷挂载到目标容器中去，也可以多次使用该参数挂载多个数据卷。下面创建一个容器，挂载一个数据卷。

```
$ sudo docker run --name=test -it -v /home/test_volume/:/home/test
[root@6a7818f6290b /]# cd /home/
[root@6a7818f6290b home]# ls
test
```

发现容器中已经成功挂载数据卷，但是如果你对系统是CentoOS7系统，你会发现，无法访问test，说明权限不够，是因为CentOS7中的安全模块selinux把权限禁掉了，所以要在运行的时候加上特权：

```
$ sudo docker run --name=test -it --privileged=true -v /home/test_v
```

这样我们就有了对容器的读写权利。（解决方法还有好多种，在后面问题总结中有所介绍。）

Docker 挂载数据卷的默认权限是读写，用户也可以通过 :ro 指定为只读。

```
$ sudo docker run -d -P --name web -v /src/webapp:/opt/webapp:ro
training/webapp python app.py
```

注意：也可以在 *Dockerfile* 中使用 *VOLUME* 来添加一个或者多个新的卷到由该镜像创建的任意容器。这将在仓库服务中详细讲解。

- ## 挂载文件

-v 标记也可以从主机挂载单个文件到容器中

```
$ sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu
```

这样就可以记录在容器输入过的命令了。

注意：如果直接挂载一个文件，很多文件编辑工具，包括 *vi* 或者 *sed --in-place*，可能会造成文件 *inode* 的改变，从 *Docker 1.1 .0*起，这会导致报错误信息。所以最简单的办法就直接挂载文件的父目录。

数据卷容器

--volumes-from 顾名思义，就是从另一个容器当中挂载容器中已经创建好的数据卷。

如果你有一些持续更新的数据需要在容器之间共享，最好创建数据卷容器。 数据卷容器，其实就是一个正常的容器，专门用来提供数据卷供其它容器挂载的。 我们首先先创建一个数据卷容器

```
$ sudo docker run -d -v /dbdata --name dbdata training/postgres ech
734273f9ec0b0cf743a79a9da7b27b269ff9c34a02ec17d3df158cf0e3cedcd2
$ sudo docker ps -a
CONTAINER ID        IMAGE                COMMAND                  CREAT
734273f9ec0b        training/postgres    "/docker-entrypoint.     17 se
```

发现创建好的数据卷容器是出于停止运行的状态，因为使用 --volumes-from 参数所挂载数据卷的容器自己并不需要保持在运行状态。

然后我们再创建容器挂载这个数据卷。

```
$ sudo docker run -d --volumes-from dbdata --name db1 training/post
$ sudo docker run -d --volumes-from dbdata --name db2 training/post
$ sudo docker ps
CONTAINER ID        IMAGE                 COMMAND                   CREA
7348cb189292        training/postgres     "/docker-entrypoint.     11 s
a262c79688e8        training/postgres     "/docker-entrypoint.     33 s
```

还可以使用多个 --volumes-from 参数来从多个容器挂载多个数据卷。 也可以从其他已经挂载了数据卷的容器来挂载数据卷。

如果删除了挂载的容器（包括 dbdata、db1 和 db2），数据卷并不会被自动删除。如果要删除一个数据卷，必须在删除最后一个还挂载着它的容器时使用 docker rm -v 命令来指定同时删除关联的容器。 这可以让用户在容器之间升级和移动数据卷。具体的操作将在下一节中进行讲解。

- 利用数据卷容器来备份、恢复、迁移数据卷

首先使用 --volumes-from 标记来创建一个加载 dbdata 容器卷的容器，并从本地主机挂载当前到容器的 /backup 目录。命令如下：

```
$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu ta
```

容器启动后，使用了 tar 命令来将 dbdata 卷备份为本地的 /backup/backup.tar。 如果要恢复数据到一个容器，首先创建一个带有数据卷的容器 dbdata2。

```
$ sudo docker run -v /dbdata --name dbdata2 ubuntu /bin/bash
```

然后创建另一个容器，挂载 dbdata2 的容器，并使用 untar 解压备份文件到挂载的容器卷中。

```
$ sudo docker run --volumes-from dbdata2 -v $(pwd):/backup busybox
/backup/backup.tar
```

Runtime constraints on CPU and memory

目前相关资料还没有收齐，还在学习之中

下面的参数可以用来调整container内的性能参数。

参数	描述
-m, --memory=""	Memory limit (format: [], where unit = b, k, m or g)
--memory-swap=""	Total memory limit (memory + swap, format: [], where unit = b, k, m or g)
-c, --cpu-shares=0	CPU shares (relative weight)
--cpu-period=0	Limit the CPU CFS (Completely Fair Scheduler) period
--cpuset-cpus=""	CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems=""	Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems.
--cpu-quota=0	Limit the CPU CFS (Completely Fair Scheduler) quota
--blkio-weight=0	Block IO weight (relative weight) accepts a weight value between 10 and 1000.
--oom-kill-disable=false	Whether to disable OOM Killer for the container or not.
--memory-swappiness=""	Tune a container’s memory swappiness behavior. Accepts an integer between 0 and 100.

通过docker run -m 可以很方便的调整container所使用的内存资源。如果host支持swap内存，那么使用-m可以设定比host物理内存还大的值。

同样，通过-c 可以调整container的cpu优先级。默认情况下，所有的container享有相同的cpu优先级和cpu调度周期。但你可以通过Docker来通知内核给予某个或某几个container更多的cpu计算周期。

默认情况下，使用-c或者--cpu-shares 参数值为0，可以赋予当前活动container 1024个cpu共享周期。这个0值可以针对活动的container进行修改来调整不同的cpu循环周期。

比如，我们使用-c或者--cpu-shares =0启动了C0，C1，C2三个container，使用-c/--cpu-shares=512启动了C3.这时，C0，C1，C2可以100%的使用CPU资源(1024)，但C3只能使用50%的CPU资源(512)。如果这个host的OS是时序调度类型

的，每个CPU时间片是100微秒，那么C0，C1，C2将完全使用掉这100微秒，而C3只能使用50微秒。

内存资源设置

设置内存我们可以有四种方式：

- **memory=inf, memory-swap=inf (default)**

  默认的方式设置最低值，容器可以使用大于此最低值的内存数

- **memory=L<inf, memory-swap=inf**

  设置*memory*不能使用超过*L*的值。

- **memory=L<inf, memory-swap=2*L**
- **memory=L<inf, memory-swap=S<inf, L<=S**

  *memory*不能超过*L*，*swap*＋*memory*总使用量不能超过*S*

例子： $ sudo docker run -ti centos /bin/bash

默认不设置任何限制。(第一种情况)

```
$ sudo docker run -ti -m 300M --memory-swap -1 centos  /bin/bash
```

memory最多使用300M，swap没有限制

```
$ docker run -ti -m 300M centos  /bin/bash
```

我们只设置了memory限制时300M，swap没有指定，默认被设置为与memory一样的值。memory＋swap一共是600M

```
$ docker run -ti -m 300M --memory-swap 1G centos  /bin/bash
```

这里我们同时设置了memory和swap，对应第四种情况

如果发生内存溢出错误，内核讲kill掉容器中的进程。如果你想控制，可以配合使用- -oom-kill-disable参数。如果没有制定-m参数，可能导致当内存溢出时内核会杀死主机进程。 例子： 设置容器内存限制100M，并且阻止 OOM killer

```
$ docker run -ti -m 100M --oom-kill-disable centos /bin/bash
```

如果不使用-m参数制定限制，官方说很危险！

CPU资源设置

默认情况下，所有容器获得CPU周期的比例相同。可以通过改变容器的CPU加权占有率相对于其他正在运行容器的加权占有率的比例来调整。

修改1024的比例，使用-c或--cpu-sharesflag的权重设置为2或更高。 该比例只适用在CPU密集型进程运行时。当在一个容器中的任务处于空闲状态，其他容器可以使用剩余空闲CPU时间。实际CPU时间将根据在系统上运行的容器的数目而变化。

例如，考虑三个容器的情况，一个拥有cpu的1024和另外两个有512 CPU共享时间，三个容器进程都尝试使用100%的CPU，第一个容器将获得的50%总的CPU时间。如果您添加CPU值为1024的第四个容器中，第一个容器只得到了CPU的33%。剩余的容器将分别占用CPU的16.5%，16.5%和33%。

在多核心系统中，CPU时间的份额分布在所有CPU核心。即使容器被限制为CPU时间小于100%时，它可以使用每个单独的CPU核心的100%。例如，在一个拥有超过三个核心的系统中，

如果启动一个容器设置-c＝512跑一个进程，另外一个设置-c=1024,跑2个进程，内存分配将会如下配置：

```
PID    container    CPU CPU share
100    {C0}      0   100% of CPU0
101    {C1}      1   100% of CPU1
102    {C1}      2   100% of CPU2
```

--cpu-period参数

默认设置为100ms，当然我们也可以自己设置cpu周期，限制容器CPU用量。通常该参数伴随--cpu-quota参数使用。

--cpu-quota参数

限制CPU用量。默认值0，意味着允许容器获得1个CPU的100%的资源量。设置50000限制CPU资源的50%。

```
$ sudo docker run -ti --cpu-period=50000 --cpu-quota=25000 centos /
```

如果是单核心系统，将意味着容器将每50ms获得50%运行周期。

--cpuset参数

设置容器允许运行的*cpu*号（在多核心系统中）：

设置容器在CPU1和CPU3上运行

```
$ sudo docker run -ti --cpuset-cpus="1,3" centos /bin/bash
```

设置容器在CPU0、CPU1、CPU2上运行

```
$ sudo docker run -ti --cpuset-cpus="0-2" centos /bin/bash
```

设置容器在指定*mems*上执行（只在*NUMA*系统中有效）：

容器只能在memory nodes 1和2上运行

```
$ sudo docker run -ti --cpuset-mems="1,3" centos /bin/bash
```

--bkio-weight参数

默认情况下，所有容器获得相同比例的blokIO带宽，这个比例值是500。要修改此比例，使用--blkio-weight设置容器的blkio相对于其他运行容器权重。它的取值范围是10～1000。 下面的例子中，设置了两个不同blkio:

```
$ sudo docker run -ti --name c1 --blkio-weight 300 centos /bin/bash
$ sudo docker run -ti --name c2 --blkio-weight 600 centos /bin/bash
```

如果同时设置两个容器blockIO，利用如下指令：

```
$ time dd if=/mnt/zerofile of=test.out bs=1M count=1024 oflag=direc
```

You'll find that the proportion of time is the same as the proportion of blkio weights of the two containers.

```
Note: The blkio weight setting is only available for direct IO. Buf
```

Docker容器访问与互联

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务。

Dockefile在网络方面除了提供一个EXPOSE之外，没有提供其它选项。下面这些参数可以覆盖Dockefile的expose默认值：

```
--expose=[]     : Expose a port or a range of ports from the containe
                  without publishing it to your host
-P=false        : Publish all exposed ports to the host interfaces
-p=[]           : Publish a container's port to the host (format:
                  ip:hostPort:containerPort | ip::containerPort |
                  hostPort:containerPort | containerPort)
                  (use 'docker port' to see the actual mapping)
--link=""       : Add link to another container (name:alias)
```

--expose可以让container接受外部传入的数据。container内监听的port不需要和外部host的port相同。比如说在container内部，一个HTTP服务监听在80端口，对应外部host的port就可能是49880。

操作人员可以使用--expose，让新的container访问到这个container。具体有三个方式

1. 使用-p来启动container。
2. 使用-P来启动container。
3. 使用--link来启动container。

如果使用-p或者-P，那么container会开发部分端口到host，只要对方可以连接到host，就可以连接到container内部。当使用-P时，docker会在host中随机一个未被占用的端口绑定到container。你可以使用docker port来查找这个随机绑定端口。

当你使用--link方式时，作为客户端的container可以通过私有网络形式访问到这个container。同时Docker会在客户端的container中设定一些环境变量来记录绑定的IP和PORT。

外部访问容器

有时候，容器要运行一些网络应用，需要外部能访问到这些应用，就需要使用-p/P参数指定一个主机端口，映射到容器端口中。其中使用P系统会分配一个随机的端口到内部容器开放的网络端口。

就拿仓库服务镜像来做例子：

```
$ sudo docker run -d -P registry
b89fc89e061dee24ac532af1890cd26e6e016545e0978b01d3d4eadca67119aa
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATEI
b89fc89e061d        registry:latest     "docker-registry"   5 secor
$ curl 192.168.4.100:32768/v1/search
{"num_results": 0, "query": "", "results": []}[root@registry liugar
$ sudo docker logs b89fc89e061d
[2015-08-18 00:11:41 +0000] [1] [INFO] Starting gunicorn 19.1.1
[2015-08-18 00:11:41 +0000] [1] [INFO] Listening at: http://0.0.0.(
```

我们可以看到，当我们加上-P时，docker会任意指定一个一个端口指定到容器的开放端口5000上。从容器到运行日志也是可以看出，在容器的5000端口会有一个监听。当我们通过外网的，也就是宿主机的IP 和端口就可以访问到该容器内提供的服务，这里是仓库服务。

-p（小写的）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。支持的格式有

```
ip:hostPort:containerPort | ip::containerPort | hostPort:containerF
```

映射所有接口地址

我们用到的是 hostPort:containerPort，也就是将制定端口映射到主机地址的任意地址的指定端口：

```
$ sudo docker run -d -p 80:5000 registry
a7abe89606427e3cb90698a6d302e8
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATEI
a7abe8960642        registry:latest     "docker-registry"   4 secor
2abb0f04066999adff36b13be2e380c3de
```

我们将主机80端口映射到容器，这样我们直接用主机地址就可以访问到容器了。

映射到指定地址的指定端口

可以使用 ip:hostPort:containerPort 格式指定映射使用一个特定地址，比如 localhost 地址 127.0.0.1

```
$ sudo docker run -d -p 127.0.0.1:5000:5000 registry
9f11390c1e9d048f7d82ff6bfb7e65f5531865343a5a2e6b660c0634e90eda26
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATEI
9f11390c1e9d        registry:latest     "docker-registry"   6 secor
```

映射到指定地址的任意端口

使用 ip::containerPort 绑定 localhost 的任意端口到容器的 5000 端口，本地主机会自动分配一个端口。

```
$ sudo sudo docker run -d -p 127.0.0.1::5000 registry
d8cd77ecc45f434ab9edc0a7e83514ef7cb019fabc9bdbc0b522bb916b309789
$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATEI
d8cd77ecc45f        registry:latest     "docker-registry"   4 secor
```

从上面看出，此时docker默认的传输协议是tcp方式，我们也可以改为其他方式标记：

```
$ sudo docker run --name=test_port -d -p 127.0.0.1:5000:5000/udp re
```

使用 docker port 查看端口信息

```
$ docker port test_port
5000/udp -> 127.0.0.1:5000
```

注意：

容器有自己的内部网络和 *ip* 地址（使用 *docker inspect* 可以获取所有的变量，*Docker* 还可以有一个可变的网络配置。）

-p 标记可以多次使用来绑定多个端口

容器间通信

容器在使用Docker的时候我们会常常碰到这么一种应用，就是我需要两个或多个容器，其中某些容器需要使用另外一些容器提供的服务。所以，我们要考虑的问题时如何建立两个容器间通信。

容器的连接（**linking**）

系统是除了端口映射外，另一种跟容器中应用交互的方式。 该系统会在源和接收容器之间创建一个隧道，接收容器可以看到源容器指定的信息。

首先我们先创建一个容器(这里我只是用作示范,没有使用官方示例的镜像，所谓但数据容器内并没有提供数据服务，官方例子我举出来也没啥意思)

创建数据访问容器db：

```
$ sudo docker run -idt --name=db centos
600886c7c69dc4979bdfee19d82331879d71835f794db110eb3b5ea3c164bd30
```

使用--link=name:alias name就是要访问的目标机器，alias就是自定的别名

```
$ sudo docker run -it --name=web --link=db:test_link centos
[root@8d92293a65e9 /]# cat /etc/hosts
172.17.0.12    8d92293a65e9
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters
172.17.0.11    test_link 600886c7c69d db
```

我们看到容器内的hosts 内多了一条信息

```
172.17.0.11    test_link 600886c7c69d db
```

这就意味着我们可以访问到db容器进行通信

高级网络配置

当docker启动时，它会在宿主机器上创建一个名为docker0的虚拟网络接口。它会从RFC 1918定义的私有地址中随机选择一个主机不用的地址和子网掩码，并将它分配给docker0。

例如我启动docker几分钟后它选择了172.17.42.1/16－一个16位的子网掩码为主机和它的容器提供了65,534个ip地址。

注意*:* 本文讨论了***Docker***的高级网络配置和选项。通常你不会用到这些。如果你想查看一个较为简单的***Docker***网络介绍和容器概念介绍来着手，请参见***Dockers***快速入门*.*

但docker0并不是正常的网络接口。它只是一个在绑定到这上面的其他网卡间自动转发数据包的虚拟以太网桥。它可以使容器与主机相互通信。每次Docker创建一个容器，它就会创建一对对等接口**(peer interface)**，类似于一个管子的两端－在这边可以收到另一边发送的数据包。Docker会将对等接口中的一个做为eth0接口连接到容器上，并使用类似于vethAQI2QT这样的惟一名称来持有另一个，该名称取决于主机的命名空间。通过将所有veth*接口绑定到docker0桥接网卡上，Docker在主机和所有Docker容器间创建一个共享的虚拟子网。

本文将会讲解使用Docker选项的所有方式，并且在高级模式下使用纯linux网线配置命令来调整，补充，或完全替代Docker的默认网络配置。

Docker选项快速指南

这里有一份关于Docker网络配置的命令行选项列表，省去您查找相关资料的麻烦。

一些网络配置的命令行选项只能在服务器启动时提供给Docker服务器。并且一旦启动起来就无法改变。

一些网络配置命令选项只能在启动时提供给Docker服务器，并且在运行中不能改变：

- **-b BRIDGE或--bridge=BRIDGE**— see 建立自己的网桥
- **--bip=CIDR**— see 定制docker0

- **-H SOCKET...或--host=SOCKET...**— 它看起来像是在设置容器的网络，但实际却恰恰相反：它告诉Docker服务器要接收命令的通道，例如“run container"和"stop container"。
- **--icc=true|false**— see 容器间通信
- **--ip=IP_ADDRESS**— see 绑定容器端口
- **--ip-forward=true|false**— see 容器间通信
- **--iptables=true|false**— see 容器间通信
- **--mtu=BYTES**— see 定制docker0

有两个网络配置选项可以在启动时或调用docker run时设置。当在启动时设置它会成为docker run的默认值:

- **--dns=IP_ADDRESS...**— see 配置DNS
- **--dns-search=DOMAIN...**— see 配置DNS

最后，一些网络配置选项只能在调用docker run时指出，因为它们要为每个容器做特定的配置:

- **-h HOSTNAME或--hostname=HOSTNAME**— see 配置DNS 和 Docker与容器连接原理
- **--link=CONTAINER_NAME:ALIAS**— see 配置DNS 和 容器间通信
- **--net=bridge|none|container:NAME_or_ID|host**— see Docker与容器连接原理
- **-p SPECor--publish=SPEC**— see 绑定容器端口
- **-P或--publish-all=true|false**— see 绑定容器端口

Docker 创建网络步骤

Docker是正在发展中的，并会持续提升网络配置的逻辑。当前命令行是很难满足Docker新建容器时所需要的网络配置。

让我们回顾一些基础知识。

通讯的时候使用网际协议（IP），一个机器需要访问至少一个网络接口用来发送和接收包，路由表定义了通过接口可达IP地址范围。网络接口不一定非是物理设备。实际上，在每一个**Linux**机器（和每个**Docker**容器内部）的**lo**回环接口都是有效的而且完全是虚拟的**——Linux**内核简单地拷贝回环（数据）包，直接从发送者的内存放入接收者的内存。

Docker使用特殊的虚拟接口让容器在主机间通讯——成对的虚拟接口被叫做“peers”，它被链接到主机内核的内部，因此（数据）包能在他们之间传输。他们简单创建，待会儿我们将会看到。

Docker配置容器的步骤是：

1. 创建一对虚拟接口
2. 在主Docker主机内部给它一个唯一的名称，比如**veth65f9**，绑定它到**docker0**或者**Docker**使用的任何网桥上
3. 让其他的接口翻墙进入新的容器（已经提供了lo接口），在容器的独立和唯一网络接口命名空间内，重新命名它为更漂亮的名字**eth0**，名称不要和其他的物理接口冲突。
4. 设置接口的MAC地址，具体使用**--mac-address** 命令指定或者随机一个。
5. 在网桥的网络地址访问内给容器的eth0一个新的IP地址，设置它的缺省路由为**Docker**主机在网桥上拥有的**IP**地址。使用--default-gateway 设置默认路由来允许该地址向Docker deamon 来转发数据，否则将使用网桥定义的IP地址来转发（docker0）。除非自定义，否则MAC地址是根据IP来生成的。当一个新的容器使用已经分配过的IP（另一个具有不同MAC地址的容器）地址启动的时候，这将可以有效防止ARP缓存失效的问题。

这些步骤结束后，容器将立即拥有一个eth0（虚拟）网卡，并会发现它自己可以和其他的容器以及互联网通讯。

你可以使用 --net= 这个选项来执行 docker run 启动一个容器，这个选项有一下可选参数。

- `--net=bridge` — 默认选项，用网桥的方式来连接docker容器。
- `--net=host` — 高数docker跳过配置容器的独立网络栈。本质上来说，这个参数告诉**docker**不去打包容器的网络层。当然，docker 容器的进程仍然被限制在它自己独有的文件系统、进程列表以及其他资源中。一个快速命令 ip addr 将像你展示docker的网络，它是建立在docker 宿主主机上的，有完整的权限去访问宿主主机的网络接口。注意这不意味着docker容器可以去重新配置宿主主机的网络栈，重新配置是需要 --privaleged=true 这个选项参数的，但是这个选项参数会让docker容器打开大量的端口以及其他的系统的超级管理权限的进程。这也会允许容器去访问宿主主机的网络服务，比如 D-bus。这会使docker容器里的进程有有权限去做一些意想不到的事，比如重启你的宿主主机。所以要谨慎使用这个选项参数。
- `--net=container:NAME_or_ID` — 告诉docker让这个新建的容器使用已有容器的网络配置。这个新建的容器将配置新的自己的文件系统和进程列表以及其他资源限制，但是将共享这个指定的容器的网络IP地址以及端口号，使得这两个容器可以通过 loopback接口相互访问。
- `--net=none` — 告诉docker为新建的容器建立一个网络栈，但不对这个网络栈进行任何配置，在这个文档的最后将介绍如何让你去建立自定义的网络配置。

去了解以下这一步是非常必要的，如果你在建立容器的时候使用 `--net=none` 这个选项参数。以下是一些命令去去配置自定义网络，就好像你让docker完全去自己配置一样。

创建一个不带任何网络配置的网络：

```
$ docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#
```

重新开启一个窗口，获取容器的pid以在 `var/run/netns/` 下便创建网络，以下会用到 `ip netns` 命令：

```
$ docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid
```

检查主机的网桥是否启用：

```
$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...
```

创建一对（peers）网络接口A和B并绑定，然后启动，其中A是在主机上，B放在容器里：

```
$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up
```

覆盖容器的网络名字空间，并将B更改为eth0，以下步骤每执行完一次都可以到第一个窗口中查看网络配置状态（ip addr）：

```
$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
$ sudo ip netns exec $pid ip link set eth0 address 12:34:56:78:9a:b
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.42.1
```

到这一步你的容器应该可以正常运行网络操作了。

当你最后退出shell以及清理掉这个容器的时候，这个容器的虚拟网络 eth0 将在网络接口A 被清除后被消除，也会自动在网桥docker0上销毁。所以不用你执行其他的命令，所有的东西将被清理。当然，是几乎所有的东西：

```
# Clean up dangling symlinks in /var/run/netns
find -L /var/run/netns -type l -delete
```

还要注意上面的脚本使用了现代的ip命令行替代旧的弃用的封装，类似**ipconfig**和**route**，些老的命令行还是会一直呆在我们的容器内部工作。如果你嫌麻烦，**ip addr**命令行也可以只键入**ip a**。

总之，注意这个**ip netns exec**重要的命令行，它让我们以**root**用户进入内部并配置一个网络命名空间。如果在容器内部运行，类似的命令行可能不会工作，因为安全容器化的部分是Docker剥离容器的处理过程，这个过程要正确地配置自己的网络。

使用 `ip netns exec` 可以让我们完成配置，还避免了运行容器自身 `--privileged=true` 的危险步骤。

定制docker0

默认地，docker服务会在linux内核新建一个网络桥接docker0，使得物理主机和其他虚拟网络接口之间可以传递发送数据包，因此，这表现如一个独立的网络。

docker0有一个IP地址和子网掩码，使得物理主机可以从容器的桥接网络接收和发送数据包。并且给这个桥接网络一个MTU（最大传输单元）或者说网络接口允许的最大包长度-例如1,500 bytes 或者从docker的宿主主机上的网络接口拷贝的数值。在服务启动的时候两者都是可配置的：

- --bip=CIDR— 为docker0桥接网络提供一个特殊的IP地址和一个子网掩码, 使用标准的 CIDR 记法例如192.168.1.5/24.
- --mtu=BYTES— 从写docker0的最大数据包长度。

在ubuntu系统上，你可以增加以上的配置到 /etc/default/docker 文件中的DOCKER_OPTS参数中，然后重启docker服务。

当你有一个或多个正常运行的容器时，你可以通过在主机上运行brct1命令，观察interfaces列的输出，来确定Docker已经将这些容器正确地连接到docker0网桥。下面是一个连接了两个不同容器的主机：

```
$ sudo brctl show
bridge name     bridge id               STP enabled       interfaces
docker0         8000.3a1d7362b4ee          no             veth65f9
                                                          vethdda6
```

最后，每次新建一个容器的时候都会用到docker0 桥接网络。每次在执行docker run命令新建一个容器的时候，docker从可利用的桥接网络中随机选择一个未被使用的IP地址，以及使用桥接网络的子网掩码，用来配置容器 eth0网络接口。docker宿主主机的IP地址被docker容器作为默认的网关。

```
$ docker run -i -t --rm base /bin/bash
[root@e623fd7cf734 /] ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state U
     link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
     inet 172.17.0.3/16 scope global eth0
           valid_lft forever preferred_lft forever
     inet6 fe80::306f:e0ff:fe35:5791/64 scope link
           valid_lft forever preferred_lft forever
[root@e623fd7cf734 /] ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
[root@e623fd7cf734 /] exit
```

建立你自己的桥接网络

如果你希望建立完整的自己的桥接网络，你可以在启动docker之前用 -b BRIDGE 或者 --bridge=BRIDGE选项参数高数docker使用你自己的桥接网络。

如果你已经用docker0启动docker了，你需要停止docker服务然后移除docker0.

```
$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
$ sudo iptables -t nat -F POSTROUTING
```

然后，在启动docker服务之前，新建你自己的桥接网络，写上你想要的配置。接下来我们新建一个简单的桥接网络，刚好用这些选项来定做docker0，这刚好足够说明这个技术。

```
$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.4.1/24 dev bridge0
$ sudo ip link set dev bridge0 up
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.4.0/24       0.0.0.0/0
```

创建好网桥之后在docker配置文件中写入启动参数，或者手动指定：

```
$ sudo docker -d --bridge=bridge0
```

这样我们自己定义的网桥就做好了，当利用docker run启动容器时，会自动绑定网络到bridge0上：

```
$ sudo docker run -it --rm centos
[root@e623fd7cf734 /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
           valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
           valid_lft forever preferred_lft forever
40: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
     link/ether 02:42:c0:a8:04:02 brd ff:ff:ff:ff:ff:ff
     inet 192.168.4.2/24 scope global eth0
           valid_lft forever preferred_lft forever
     inet6 fe80::42:c0ff:fea8:402/64 scope link
           valid_lft forever preferred_lft forever
```

我们可以看到新的网桥已经生效。

容器与外部网络通信

决定容器是否可以访问外网取决于两个因素：

1. 主机是否会转发IP数据包。这取决于转发系统内的ip_forward这个参数的配置。如果ip_forward值为1，数据包就可以被转发。Docker会使用--ip_forward=true的默认设置，一旦你docker服务启动docker会将系统的ip_forward的值修改为1。使用-ip_forward=false对系统没有改变。通常设置方法如下：

   ```
   $ sysctl net.ipv4.conf.all.forwarding
   net.ipv4.conf.all.forwarding = 0
   $ sysctl net.ipv4.conf.all.forwarding=1
   $ sysctl net.ipv4.conf.all.forwarding
   net.ipv4.conf.all.forwarding = 1
   ```

   设置这个值很重要，它将决定你的容器是否可以与外部通信以及容器间的通信。在多网桥系统中，内嵌的容器也需要配置此项。

2. 你的iptables防火墙是否允许特殊连接。当deamon启动的时候，如果你设置--iptables=false，Docker将不会改变主机的iptables的防火墙规则。否则，Docker将追加规则到DOCKER filter chain

   Docker 不会删除或修改已经存在在DOCKER filter chain中的规则。这允许用户进一步创建限制访问容器的任何规则。

   Docker的默认规则是允许所有的外部IPs。如果只是想允许一个IP连接到容器，在DOCKER filter chain的顶部增加一条否定规则：

   ```
   $ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
   ```

   这将只允许 8.8.8.8这个IP连接到容器。

容器间互相访问

决定容器能否互相访在系统层面上问取决于两个因素：

1. 网络的拓扑结构是否已经连接到容器的网络接口。默认情况下，Docker会把所有的容器附加到docker0网桥下，并为两个容器间的包传输提供路径。

2. iptables是否允许特殊连接?如果你把设置 --iptables=false,当守护进程启动时，Docker不会改变你的系统iptables规则。另外，如果你保留默认设置 --icc=true，Docker服务器或向FORWARD链添加一个带有全局ACCEPT策略的默认规则。如果不保留默认设置即--icc=false，系统会把策略设为DROP.

   使用docker都希望ip_forward 是打开的，至少使容器间的通讯成为可能。

   但是否同意 --icc=true 或者更改为 --icc=false 使得iptables 可以保护容器以及宿主主机不被任意地端口扫描、避免被已经被渗透的容器所访问，这是一个策略问题。（在ubuntu，是编辑/etc/default/docker文件中的DOCKER_OPTS参数，然后重启docker服务）

如果你选择最安全的设置 **--icc=false**，那么当你想让它们彼此提供服务的时候如何让它们相互通讯？

答案是：使用前文提到的 --link=CONTAINER_NAME:ALIAS 选项。如果docker守护进程正在以 --icc=false 和 --iptables=true 参数运行，当以选项 --link= 执行 docker run 命令时，docker服务将插入一部分 iptables ACCEPT 规则使得新容器可以连接其他容器所暴露出来的端口（此端口指前文在 Dockerfile 中提到的EXPOSE 这一行）。更多详细文档介绍请看：容器互联。

注意: --link 选项中的 CONTAINER_NAME 的值必须是 docker自动分配的容器名称，比如stupefied_pare，或者是在执行docker run 的时候用--name=指定的容器名称. 这不能使一个docker无法识别的主机名

你可以在你的Docker主机上运行iptables命令，来观察FORWARD链是否有默认的ACCEPT或DROP策略

```
# When --icc=false, you should see a DROP rule:
$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...
# When a --link= has been created under --icc=false,
# you should see port-specific ACCEPT rules overriding
# the subsequent DROP policy for all other packets:
$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tcp
ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tcp
```

配置DNS

怎样为Docker提供的每一个容器进行主机名和DNS配置，而不必建立自定义镜像并将主机名写 到里面？它的诀窍是覆盖三个至关重要的在/etc下的容器内的虚拟文件，那几个文件可以写入 新的信息。你可以在容器内部运行mount看到这个：

```
$$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/resolv.conf type ext4 ...
...
```

HCP配置之后，保持resolv.conf的数据到所有的容器中。Docker怎样维护在容器内的这些文件从Docker的一个版本到下一个版本的具体细节，你应该抛开这些单独的文件本身并且使用下面的Docker选项代替。

有四种不同的选项会影响容器守护进程的服务名称。

1. -h HOSTNAME 或 --hostname=HOSTNAME --设置容器的主机名，仅本机可见。这种方式是写到/etc/hostname ，以及/etc/hosts 文件中，作为容器主机IP的别名，并且将显示在容器的bash中。不过这种方式设置的主机名将不容易被容器之外可见。这将不会出现在 docker ps 或者 其他的容器的/etc/hosts 文件中。

   ```
   $ sudo docker run --hostname 'myhost' -it centos
   [root@myhost /]# cat /etc/hosts
   172.17.0.7    myhost
   …
   ```

2. --link=CONTAINER_NAME:ALIAS --使用这个选项去run一个容器将在此容器的/etc/hosts文件中增加一个主机名ALIAS，这个主机名是名为CONTAINER_NAME 的容器的IP地址的别名。这使得新容器的内部进程可以访问主机名为ALIAS的容器而不用知道它的IP。--link= 关于这个选项的详细讨论请看：容器互联

3. --dns=IP_ADDRESS --设置DNS服务器的IP地址，写入到容器的/etc/resolv.conf文件中。当容器中的进程尝试访问不在/etc/hosts文件中的主机A时，容器将以53端口连接到IP_ADDRESS这个DNS服务器去搜寻主机A的IP地址。

```
$  sudo docker run -it --dns=192.168.5.1  centos
[root@6a38049c9052 /]# cat /etc/resolv.conf
nameserver 192.168.5.1
```

4. --dns-search=DOMAIN --设置DNS服务器的搜索域，以防容器尝试访问不完整的主机名时从中检索相应的IP。这是写入到容器的/etc/resolv.conf文件中的。当容器尝试访问主机 host，而DNS搜索域被设置为 example.com ,那么DNS将不仅去查寻host主机的IP，还去查询host.example.com的IP。

```
$  sudo docker run -it --dns-search=www.domain.com  centos
[root@ae0e9e99596f /]# cat /etc/resolv.conf
nameserver 192.168.4.1
search www.mydomain.com
```

在docker中，如果启动容器时缺少以上最后两种选项设置时，将使得容器的/etc/resolv.conf文件看起来和宿主主机的**/etc/resolv.conf**文件一致。这些选项将修改默认的设置。(本宿主机在实验时有一行“nameserver 192.168.4.1”，所以默认容器的配置会与宿主机一样。)

为主机绑定容器端口

默认情况下，Docker容器可以连接到外部区域，但外部区域不能连接到容器。在Docker启动时，由于它在主机上创建了一个**iptables**伪装规则，使得每一个输出连接看起来都是由主机IP地址建立起来的。

```
# You can see that the Docker server creates a
# masquerade rule that let containers connect
# to IP addresses in the outside world:
$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       0.0.0.0/0
...
```

当调用docker run的时候，如果你想让容器接受输入连接，你需要提供特殊选项。这些选项的详细说明在 Docker 快速入门. 有两种方法可以实现。

首先，你可以提供 **-P** 或者**--publish-all=true|false** 选项参数来执行 docker run 命令，这将会识别所有在dockerfile中暴露的端口或者--expose参数制定的端口，然后用检查端口映射，临时端口范围由/proc/sys/net/ipv4/ip_local_port_range内核参数配置，并且随机映射到 **32768** ～ **61000** 之间的主机端口。

更方便的操作是使用 **-p SPEC** 或者**--publish=SPEC** 选项，这两个选项让你明确的指定docker容器的端口映射到任意的主机端口中，不局限于32768 ～ 61000.

无论如何，你应该通过审查你的NAT表，去看看docker在你的网络占做了什么。

```
# What your NAT rules might look like when Docker
# is finished setting up a -P forward:
$ iptables -t nat -L -n
...
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp d
# What your NAT rules might look like when Docker
# is finished setting up a -p 80:80 forward:
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp d
```

可以看到，docker暴露了这些容器的端口到通配IP地址：0.0.0.0， 这个通配IP地址可以匹配宿主主机上任意一个可以进入的端口。如果你希望更多的限制，并且只允许容器服务通过特殊的宿主主机的外部网络接口来相互联系，那么你有两种选择。当你执行 docker run 命令时，你可以使用 **-p IP:host_port:container_port** 或者 **-p IP::port**来明确地绑定外部接口。

或者如果你希望dokcer永远转发到一个特殊的IP地址上，你可以编辑你的docker系统设置文件（ubuntu系统的设置方法为：编辑 /etc/default/docker文件，改写 DOCKER_OPTS参数），增加选项 --ip=IP_ADDRESS 。修改完之后记得重启你的**docker**服务。

如果你希望更详细的指导，请参考：Docker 快速入门.

官方创建overlay网络

Step 1: Set up a key-value store

overlay网络需要key－value存储。存储内包含一些网络状态：discovery, networks, endpoints, ip-addresses等。docker支持Consul，Etcd以及Zookeeper（分布式存储）key－value 存储。以下以Consul为例。

1.登陆安装好以下软件等系统：

```
Docker Engine, Docker Machine, and VirtualBox software.
```

2.创建一个名叫 mh-keystore 的machine.

```
$ docker-machine create -d virtualbox mh-keystore
```

当你配置好了一个machine, 该进程将给主机添加Docker Engine。这意味着你无需手动安装Consual，而是使用Docker HUB上的Consual镜像来创建。下一步将执导你怎么做。

3.在 mh-keystore machine 上启动 progrium/consul 容器.

```
$  docker $(docker-machine config mh-keystore) run -d \
-p "8500:8500" \
-h "consul" \
progrium/consul -server -bootstrap
```

该命令将启动aprogrium/consul 镜像在mh-keystore machine上创建容器,
consuland监听端口是8500.

4.给 mh-keystore machine设置本地的环境变量.

```
$  eval "$(docker-machine env mh-keystore)"
```

5.使用docker run 查看consul container.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CI
4d51392253b3        progrium/consul     "/bin/start -server -"   25
```

Step 2: Create a Swarm cluster

在该步骤, 你将使用 docker-machine 来扩展主机网络. 针对这点， 你不是真的创建一个网络. 而是在VirtualBox中创建了一些machines .其中一个 machine 将被当作Swarm master; 首先你得创建它. 当你创建完所有主机之后,你要将overlay网络驱动需要的配置参数传递给Engine。

1.创建Swarm master.

```
$ docker-machine create \
-d virtualbox \
--swarm --swarm-image="swarm" --swarm-master \
--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500"
--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystor
--engine-opt="cluster-advertise=eth1:2376" \
mhs-demo0
```

在创建期间, 你要给 Engine daemon提供 --cluster-store 参数. 这个参数告诉Engine ，overlay 网络所需要的Key－value存储的位置。脚本$(docker-machine ip mh-keystore) 解析出你在第一步中创建的Consul server的IP address 。--cluster-advertise 参数advertises 将在网络上暴露 machine 。

2.创建其它主机加入到集群中.

```
$ docker-machine create -d virtualbox \
--swarm --swarm-image="swarm:1.0.0-rc2" \
--swarm-discovery="consul://$(docker-machine ip mh-keystore):8500"
--engine-opt="cluster-store=consul://$(docker-machine ip mh-keystor
--engine-opt="cluster-advertise=eth1:2376" \
mhs-demo1
```

3.列出machines来验证集群是否启动。

```
$ docker-machine ls
NAME          ACTIVE   DRIVER       STATE     URL
default                virtualbox   Running   tcp://192.168.99.100:2
mh-keystore            virtualbox   Running   tcp://192.168.99.103:2
mhs-demo0              virtualbox   Running   tcp://192.168.99.104:2
mhs-demo1              virtualbox   Running   tcp://192.168.99.105:2
```

此时你已经有了运行在网络上的主机，我们准备利用这些主机为容器创建多主机网络。保持终端连接来进行下一步。

Step 3: Create the overlay Network

创建 overlay 网络

1.为 Swarm master设置docker环境。

```
$ eval $(docker-machine env --swarm mhs-demo0)
```

使用 --swarm 参数 with docker-machine 限制 the dockercommands to Swarm information alone.

2.使用docker info 查看Swarm.

```
$ docker info
Containers: 3
Images: 2
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
mhs-demo0: 192.168.99.104:2376
└ Containers: 2
└ Reserved CPUs: 0 / 1
└ Reserved Memory: 0 B / 1.021 GiB
└ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2doc
mhs-demo1: 192.168.99.105:2376
└ Containers: 1
└ Reserved CPUs: 0 / 1
└ Reserved Memory: 0 B / 1.021 GiB
└ Labels: executiondriver=native-0.2, kernelversion=4.1.10-boot2doc
CPUs: 2
Total Memory: 2.043 GiB
Name: 30438ece0915
从以上信息看出，在Master运行了3个容器以及保存了2个镜像。
```

3.创建 overlay 网络.

```
$ docker network create --driver overlay my-net
```

你只需在其中一台机器上创建一个overlay的网络即可.在本例中，我们使用master创建了一个overlay网络，但是你也可以在集群的其它任何主机上使用它。

4.验证网络是否可用:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
412c2496d0eb        mhs-demo1/host      host
dd51763e6dd2        mhs-demo0/bridge    bridge
6b07d0be843f        my-net              overlay
b4234109bd9b        mhs-demo0/none      null
1aeead6dd890        mhs-demo0/host      host
d0bb78cbe7bd        mhs-demo1/bridge    bridge
1c0eb8f69ebb        mhs-demo1/none      null
```

由于你处于Swarm master环境, 你可以看到在所有的 Swarm 代理上的所有网络. 注意每一个 NETWORK ID 唯一的. 默认的网络就是 overlay network.

5.依次切换到each Swarm 代理，然后列出网络

```
$ eval $(docker-machine env mhs-demo0)
$ docker network ls
NETWORK ID          NAME                DRIVER
6b07d0be843f        my-net              overlay
dd51763e6dd2        bridge              bridge
b4234109bd9b        none                null
1aeead6dd890        host                host

$ eval $(docker-machine env mhs-demo1)
$ docker network ls
NETWORK ID          NAME                DRIVER
d0bb78cbe7bd        bridge              bridge
1c0eb8f69ebb        none                null
412c2496d0eb        host                host
6b07d0be843f        my-net              overlay
```

所有的节点都展示出它们有ID为 6b07d0be843f 名为my-net的网络 。此时你的overlay网络就运行成功了。

手动搭建overlay网络

环境配置

配置三台机器(注意一定要修改hostname) :

```
consul:         192.168.31.181
node1:          192.168.31.182
node2:          192.168.31.183
```

三台机器同时都要安装*docker1.9*

关闭防火墙

将内核升级到*3.18,*这里升级到了*4.3*

STEP1. 选择consul主机， 启动consul服务，这里启动的是一个单节点的Consul服务，使用docker提供的容器服务可以方便的让我们启动它，只需简单输入：

```
docker run -d -p "8500:8500" -h "consul" progrium/consul -server -b
```

STEP2. 在node节点上启动docker daemon 进程：

```
docker daemon -D -g /var/lib/docker -H unix:// -H tcp://0.0.0.0:237
```

--cluster-store指定了consul服务发现地址，--cluster-advertise指定了本机服务注册地址，其中enp0s3是自己网卡的名称，也可以用内网网址192.168.31.182代替。

STEP3.启动好了之后一开始daemon会no route类似的错误，不用管它，过一会就会注册好，就会报如下类似信息：

```
DEBU[6143] Watch triggered with 2 nodes                  discovery=
DEBU[6148] Watch triggered with 2 nodes                  discovery=
DEBU[6148] Watch triggered with 2 nodes                  discovery=
DEBU[6149] 2015/11/25 02:08:12 [DEBUG] memberlist: Initiating push/
```

memberlist 说明可以发现其它节点，这就说明已经注册成功，也可打开consul所在的主机ip：8500查看：

```
http://192.168.31.181:8500
```

可以看到consul的ui界面。

STEP4. 使用docker network命令：

```
docker network ls

docker network create --driver overlay my-net

docker run -it --net=my-net  centos
```

Docker 私有仓库

如果你想玩转docker，一个私有仓库是必不可少的。 本文将会搭建一个简易的私有仓库以供参考。

本文例子的主机地址是

```
192.168.4.121
```

第一步 获取官方工具

官方为我们提供了一个创建仓库的工具，它是以镜像文件形式存储在官方仓库中，我们可以把它拉下来用。

```
$ sudo docker pull registry
```

第二步 启动仓库

我们现在启动它,指定主机5000端口绑定

```
$ sudo docker -d  -p 5000:5000 registry
```

第三步 验证

这时 输入

```
$ curl http://192.168.4.160:5000/v1/search
{"num_results": 0, "query": "", "results": [] }
```

产生如上结果说明仓库服务可用。

至此，仓库的简易配置就结束了。

但是问题来了，我们pull，push的文件在哪？哈哈，原来默认情况下，会将仓库存放于容器内的/tmp/registry目录下，这样如果容器被删除，则存放于容器中的镜像也会丢失，所以我们一般情况下会指定本地一个目录挂载到容器内的/tmp/registry下：

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/regist
```

我们将本机的centos镜像tag一下

```
$ sudo docker tag centos 192.168.4.160:5000/centos:latest
```

这时将本地仓库push到私有仓库中去

```
$ sudo docker push 192.168.4.160:5000/centos
```

可是失败了

```
2015/08/05 11:01:17 Error: Invalid registry endpoint https://192.16
```

这是因为docker采用了安全机制，若想跳过此安全验证，可以在docker配置文件中添加--insecure-registry 192.168.4.160:5000该参数标记该仓库允许不安全连接，这在deamon中提到过。

修改/etc/sysconfig/docker文件

```
OPTIONS='--selinux-enabled --insecure-registry 192.168.4.160:5000'
```

这样应该就可以了。

```
$ sudo docker push 192.168.4.160:5000/progrium
The push refers to a repository [192.168.4.160:5000/progrium] (len
Sending image list
Pushing repository 192.168.4.160:5000/progrium (1 tags)
511136ea3c5a: Image successfully pushed
d7ac5e4f1812: Image successfully pushed
2f4b4d6a4a06: Image successfully pushed
83ff768040a0: Image successfully pushed
6c37f792ddac: Image successfully pushed
e54ca5efa2e9: Image successfully pushed
2d07e6ffe5ad: Image successfully pushed
a2de3cd83939: Image successfully pushed
8d2c32294d38: Image successfully pushed
873c28292d23: Image successfully pushed
Pushing tag for rev [873c28292d23] on {http://192.168.4.160:5000/v1
```

查看仓库内容

```
$ curl 192.168.4.100/v1/search
{"num_results": 5, "query": "", "results": [{"description": null, '
```

说明成功了！！

单主机 Standalone 模式

分析：zookeepr在单节点上是以Standalone方式运行，想必在看过本文之前读者也试验过，所以我这里也不会详细说明。其实区分单节点或多节点集群zookeeper最核心大配置就是zoo.cfg，若zoo.cfg文件中包含server,那么在启动zookeeper服务的时候就会以集群方式运行。下面列出单节点zoo.cfg的详细内容：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/tmp/zookeeper
clientPort=2181
```

在下载好的zookeeper中的conf文件夹下有一个zoo_sample.cfg,我们直接将其改为zoo.cfg即可。这里，我们会用2181端口来访问zookeeper服务。

接下来我们来创建zookeeper的镜像。

构建Dockerfile zookeeper需要安装配有java环境，因此，需要将java环境配置进去：

```
ADD jdk/  /var/jdk/
ENV JAVA_HOME  /var/jdk
ENV PATH       $JAVA_HOME/bin:$PATH
```

我们再将zookeeper添加进去，并暴露2181端口：

```
ENV ZK_HOME    /var/zookeeper
EXPOSE 2181
```

这样就可以了，具体dockerfile文件内容如下：

```
# CentOS as base image
# version 0.3
# Author: liugang

# Base image to use, this must be set as the first line
FROM centos

MAINTAINER liugang 18936615621@e189.com

#ADD JAVA ZOOKEEPER
ADD zookeeper.tar /var
ADD jdk/  /var/jdk/

#ADD ENV
ENV JAVA_HOME  /var/jdk
ENV ZK_HOME    /var/zookeeper
ENV PATH       $JAVA_HOME/bin:$PATH

# Commands to update the image
RUN source /etc/profile

#Expose
EXPOSE 2181 2888 3888

ENTRYPOINT ["/var/zookeeper/bin/zkServer.sh"]

CMD ["start-foreground"]
```

以上Dockerfile制作的镜像在集群下也可以使用。

构建镜像：

```
# docker build -t="zookeeper" .
Sending build context to Docker daemon 348.1 MB
Sending build context to Docker daemon
Step 0 : FROM centos
 ---> a8415304daac
Step 1 : MAINTAINER liugang 18936615621@e189.com
 ---> Running in 912028e3b1d4
```

```
 ---> e7574b19226e
Removing intermediate container 912028e3b1d4
Step 2 : ADD zookeeper.tar /var
 ---> e956ed3cb80e
Removing intermediate container 9e1954b57eeb
Step 3 : ADD jdk/ /var/jdk/
 ---> 327fede9154e
Removing intermediate container cb3551ca5abc
Step 4 : ENV JAVA_HOME /var/jdk
 ---> Running in 2070662c6cad
 ---> 395b3fab1a07
Removing intermediate container 2070662c6cad
Step 5 : ENV ZK_HOME /var/zookeeper
 ---> Running in d535dffc8faf
 ---> d05e2d002c45
Removing intermediate container d535dffc8faf
Step 6 : ENV PATH $JAVA_HOME/bin:$PATH
 ---> Running in c9940b535a9d
 ---> 494ed0d313e1
Removing intermediate container c9940b535a9d
Step 7 : RUN source /etc/profile
 ---> Running in 4477bc347f82
 ---> 040be16683cc
Removing intermediate container 4477bc347f82
Step 8 : RUN mv /var/zookeeper/conf/zoo_sample.cfg /var/zookeeper/c
 ---> Running in e8480d49b717
 ---> b21de36d6251
Removing intermediate container e8480d49b717
Step 9 : EXPOSE 2181 2888 3888
 ---> Running in a840e7759057
 ---> c4733cfba9d1
Removing intermediate container a840e7759057
Step 10 : ENTRYPOINT /var/zookeeper/bin/zkServer.sh
 ---> Running in 6d6ec643f542
 ---> b4cdf9db68cc
Removing intermediate container 6d6ec643f542
Step 11 : CMD start-foreground
 ---> Running in 2d3eedc2c196
 ---> df307fb18230
Removing intermediate container 2d3eedc2c196
```

```
Successfully built df307fb18230
```

接下来测试，我们利用创建好的容器镜像运行一个容器：

```
# docker  run -d -P --name test zookeeper
620bde126a2f945e62de49adf7592ded71eec86041e441cb618bca198b9d9d55
# docker ps
CONTAINER ID        IMAGE               COMMAND                CREA
620bde126a2f        zookeeper           "/var/zookeeper/bin/   6 se
```

已经运行成功，我们可以通过telnet测试：

```
# telnet 0.0.0.0 32773
Trying 0.0.0.0...
Connected to 0.0.0.0.
Escape character is '^]'.
stat
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Clients:
 /172.17.42.1:53747[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
```

可以看到，在容器内部，zookeeper是以standalone方式运行的。

单主机集群容器模式

单节点（主机）的zookeeper容器搭建，原理也是比较简单的，我们利用之前创建的zookeeper镜像分别创建三个容器：

Zk1

```
sudo docker run -d -p 21811:2181 --name=container1 \
-v  /home/zk/container1:/var/zookeeper/data \
-v  /home/zk/zoo.cfg:/var/zookeeper/conf/zoo.cfg \
-v  /home/zk/container_exec.sh:/var/zookeeper/data/container_exec.s
zookeeper /var/zookeeper/data/container_exec.sh
```

Zk2

```
sudo docker run -d -p 21812:2181 --name=container3 \
-v  /home/zk/container1:/var/zookeeper/data \
-v  /home/zk/zoo.cfg:/var/zookeeper/conf/zoo.cfg \
-v  /home/zk/container_exec.sh:/var/zookeeper/data/container_exec.s
zookeeper /var/zookeeper/data/container_exec.sh
```

Zk3

```
sudo docker run -d -p 21813:2181 --name=container3 \
-v  /home/zk/container1:/var/zookeeper/data \
-v  /home/zk/zoo.cfg:/var/zookeeper/conf/zoo.cfg \
-v  /home/zk/container_exec.sh:/var/zookeeper/data/container_exec.s
zookeeper /var/zookeeper/data/container_exec.sh
```

其中container_exec.sh负责在容器中配置zoo.cfg文件、获取myid文件内容以及启动容器操作，具体内容如下：

```
#获取容器ip地址，以及myid文件
CONTAINER_IP=`ip addr | grep eth0 |grep inet|cut -d/ -f1|awk '{prin
ZK_ID=`cat /var/zookeeper/data/myid`

#配置zoo.cfg文件
echo $CONTAINER_IP
echo myid=$ZK_ID
echo 'server.'$ZK_ID'='$CONTAINER_IP':2888:3888' >> /var/zookeeper/

#等待其他容器配置好
sleep 5

cat /var/zookeeper/conf/zoo.cfg

#前台运行
/var/zookeeper/bin/zkServer.sh start-foreground
```

待我们运行好了之后，我们可以看到端口映射：

```
# docker ps
CONTAINER ID        IMAGE               COMMAND                CREA
b17517a0765e        zookeeper           "/var/zookeeper/data   8 se
b04153b0197b        zookeeper           "/var/zookeeper/data   9 se
17c260876404        zookeeper           "/var/zookeeper/data   9 se
```

然后查看zookeper集群状态：

```
# telnet 0.0.0.0 21812
Trying 0.0.0.0...
Connected to 0.0.0.0.
Escape character is '^]'.
stat
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Clients:
 /172.17.42.1:57481[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x100000000
Mode: leader
Node count: 4
Connection closed by foreign host.
```

至此搭建成功

跨主机集群容器模式

在同一台主机上运行多个zookeeper容器可以实现集群方式，并且可以很方便的文件共享（数据卷）。

但是遇到跨主机访问就不会那么方便，容器的跨主机网络的访问官方没有提供现成的方案。

不过官方提供的高级网络配置中，可以利用其配置原理，自己搭建一个网桥，实现容器的互相访问，**pipework**就是这样实现了跨主机访问，有兴趣的话可以关注一下，这不是本文的重点。

本文将结合之前试验过的 `Open vSwitch` 的网络虚拟交换机技术来实现跨主机访问，这里只是使用，详细信息请点这里。

配置Open vSwitch 环境

在配置zookeeper之前，需要配置OVS虚拟交换机，我们可以运行脚本OVSconf，就可以完成配置。需要注意的是，该脚本需要对网络的访问支持。运行完该脚本之后就可以使用Open vSwitch了（脚本之在CentOS7下配置，需要root权限，可能运行中会出现问题，这时请与我联系。）完成后，查看服务是否启动正确，

```
# systemctl status openvswitch -l
```

再查看接口状态：

```
# ovs-vsctl show
49947409-269d-44d3-a851-3927d89e1968
    Bridge "ovs0"
        Port "ovs0"
            Interface "ovs0"
                type: internal
        Port "docker0"
            Interface "docker0"
        Port "eno16777736"
            Interface "eno16777736"
    ovs_version: "2.4.0"
```

我们看到，docker0已经成功添加到网桥ovs0上了。

配置zookeeper文件

本次试验主机操作系统为CentOS7的两台虚拟主机，主机的IP地址为：

```
Server1:192.168.4.121
Server2:192.168.4.123
```

创建容器：

```
Server1:

    Docker deamon配置为：

    # docker -d --fixed-cidr=172.17.0.0/24

    运行容器并分别进入容器（需打开多个shell窗口）：

    # docker run -it --name=zk1 zookeeper
    # docker run -it --name=zk2 zookeeper

Server2:

    Docker deamon配置为：

    # docker -d --fixed-cidr=172.17.1.0/24

    运行容器并分别进入容器：

    # docker run -it --name=zk3 zookeeper
```

进入容器后，可以使用ip addr命令查看docker0分配的ip：

```
zk1 172.17.0.1
zk2 172.17.0.1
zk3 172.17.0.1
```

故进入容器内部分别配置zoo.cfg，如下：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/zookeeper/data
clientPort=2181
server.1=172.17.0.1:2888:3888
server.2=172.17.0.2:2888:3888
server.3=172.17.1.1:2888:3888
```

配置好之后，我们在容器内部运行:

zk1:

```
[root@e3c26bffd1d0 /]# mkdir /var/zookeeper/data
[root@e3c26bffd1d0 /]# vi /var/zookeeper/data/myid
[root@e3c26bffd1d0 /]# echo 1 > /var/zookeeper/data/myid
[root@e3c26bffd1d0 /]# /var/zookeeper/bin/zkServer.sh start-foregro
```

zk2

```
[root@b60565e62c2f /]# mkdir /var/zookeeper/data
[root@b60565e62c2f /]# vi /var/zookeeper/data/myid
[root@b60565e62c2f /]#  echo 2 > /var/zookeeper/data/myid
[root@b60565e62c2f /]#  /var/zookeeper/bin/zkServer.sh start-foregr
```

zk3

```
[root@1d8cb4d6663d /]# mkdir /var/zookeeper/data
[root@1d8cb4d6663d /]# vi /var/zookeeper/data/myid
[root@1d8cb4d6663d /]# echo 3 > /var/zookeeper/data/myid
[root@1d8cb4d6663d /]# /var/zookeeper/bin/zkServer.sh start-foregro
```

接下来我们就运行，进入容器，到zookeeper/bin目录下，zkServer.sh start（也可以start-foreground）。

测试

到主机上测试：

我们没有做端口映射，这样主机是访问不了容器的，我们可以利用ovs给容器添加一个192.168.4.0网段的ip：

```
# ovs-docker add-port ovs0 eth1 zk1 --ipaddress=192.168.4.125/24 --
# telnet 192.168.4.125 2181
Trying 192.168.4.125...
Connected to 192.168.4.125.
Escape character is '^]'.
stat
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Clients:
 /192.168.4.121:39150[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x100000000
Mode: follower
Node count: 4
Connection closed by foreign host.
```

可以看到跨主机的zookeeper配置已经完成，本文简单介绍了OVS的用法，更高级的网络配置请关注后续更新。